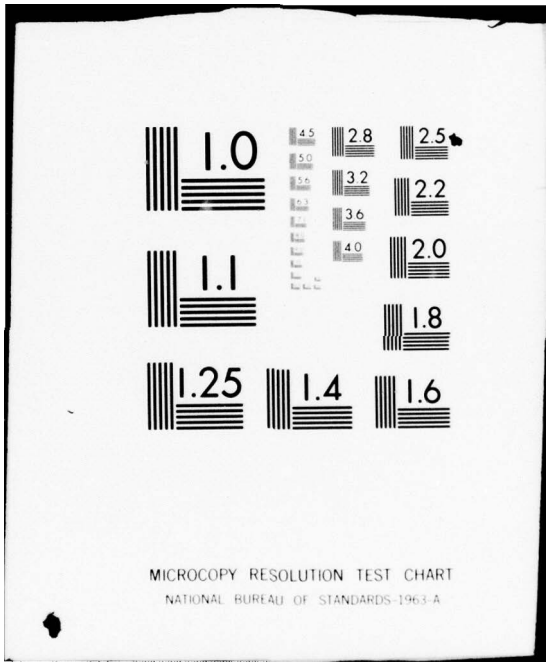
AD-A057 646 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 5/1
COMPUTER SYSTEM DESIGN USING A HIERARCHICAL APPROACH TO PERFORM--ETC(U)
OCT 77 B KUMAR DAAB07-72-C-0259
UNCLASSIFIED R-799 NL

1 OF 2
AD A057646


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LEVEL II

REPORT R-799 OCTOBER, 1977 UILU-ENG 77-2246

CSL COORDINATED SCIENCE LABORATORY

COMPUTER SYSTEM DESIGN USING A HIERARCHICAL APPROACH TO PERFORMANCE EVALUATION

BALASUBRAMANIAN KUMAR

78 08 15 027

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

COMPUTER SYSTEM DESIGN USING A HIERARCHICAL APPROACH TO PERFORMANCE EVALUATION

5. TYPE OF REPORT & PERIOD COVERED

Technical Report

6. PERFORMING ORG. REPORT NUMBER

R-799; UILU-ENG 77-2246

7. AUTHOR(s)

Balasubramanian Kumar

8. CONTRACT OR GRANT NUMBER(s)

DAAB-07-72-C-0259;
MCS 73-03488 A01

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Joint Services Electronics Program

12. REPORT DATE

October, 1977

13. NUMBER OF PAGES

161

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Design
Hierarchy
Modeling
Performance Evaluation
Optimization

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The concept of a hierarchy of system models for the performance evaluation of computer systems is introduced. The characteristics and construction of such a hierarchy are discussed. Since it consists of models that span a wide range of complexity and cost, such a hierarchy is a very useful tool in the cost-effective design of computer systems.

A procedure that uses such a hierarchy in computer system design is developed. The procedure uses the hierarchy to trade off cost and accuracy of system performance predictions. The viability and usefulness of the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

procedure are demonstrated by applying it to the optimization of the architecture of a complex computer system - the CPU-memory subsystem of the IBM System 360/Model 91. In most of the experiments conducted, the procedure converged, if not to the exact optimum, at least to within a very near region of the optimum. A sensitivity analysis procedure is then used to identify the exact optimum, as well as to determine the sensitivity of the objective function to changes in the system parameters. The efficiency of the overall procedure is shown to be considerably greater than that of the worst-case approach to system design.

Some conclusions are drawn about this class of single stream, highly pipelines, CPU-memory architectures. Extensions of the hierarchical approach to performance evaluation are proposed.

LEVEL II

(12)

(14) R-799, UILU-ENG-77-2246

(6) COMPUTER SYSTEM DESIGN USING A HIERARCHICAL APPROACH TO PERFORMANCE EVALUATION,

by

(10) BALASUBRAMANIAN/KUMAR

(9) Doctoral thesis,

(11) Oct 77 (12) 169 p.

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259 and in part by the National Science Foundation Grant under MCS 73-03488 A01.

(15) DAAB07-72-C-0259, NSF-MCS73-03488

Approved for public release. Distribution unlimited.

097 700

COMPUTER SYSTEM DESIGN USING A HIERARCHICAL APPROACH TO PERFORMANCE EVALUATION

BY

BALASUBRAMANIAN KUMAR

B.Tech., Indian Institute of Technology, 1973
M.S., University of Illinois, 1976

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1978

Thesis Adviser: Professor Edward S. Davidson

Urbana, Illinois

COMPUTER SYSTEM DESIGN USING A HIERARCHICAL APPROACH TO PERFORMANCE EVALUATION

Balasubramanian Kumar, Ph.D.
Coordinated Science Laboratory and
Department of Computer Science
University of Illinois at Urbana-Champaign, 1978

The concept of a hierarchy of system models for the performance evaluation of computer systems is introduced. The characteristics and construction of such a hierarchy are discussed. Since it consists of models that span a wide range of complexity and cost, such a hierarchy is a very useful tool in the cost-effective design of computer systems.

A procedure that uses such a hierarchy in computer system design is developed. The procedure uses the hierarchy to trade off cost and accuracy of system performance predictions. The viability and usefulness of the procedure are demonstrated by applying it to the optimization of the architecture of a complex computer system - the CPU-memory sybsystem of the IBM System 360/Model 91. In most of the experiments conducted, the procedure converged, if not to the exact optimum, at least to within a very near region of the optimum. A sensitivity analysis procedure is then used to identify the exact optimum, as well as to determine the sensitivity of the objective function to changes in the system parameters. The efficiency of the overall procedure is shown to be considerably greater than that of the worst-case approach to system design.

Some conclusions are drawn about this class of single stream, highly pipelines, CPU-memory architectures. Extensions of the hierarchical approach to performance evaluation are proposed.

ACKNOWLEDGMENT

The author is struck by a strange feeling of inadequacy in trying to acknowledge Professor Ed Davidson's contribution to this work. Professor Davidson's guidance, friendship, encouragement and genuine personal warmth render conventional expressions of gratitude woefully inadequate. The author can only hope that some day, he himself will be able to contribute so much to someone else's work.

The author thanks all his colleagues and professors in the Department of Computer Science and the Coordinated Science Laboratory, especially Professor Dave Kuck, Janak Patel, Ravi Nair, Joel Emer, Trevor Mudge and Alan Gant for contributing so much to his education.

For making his stay in Urbana-Champaign so much fun, the author is deeply indebted to his friends Arvind and Sashi Parthasarathi.

Finally, the author thanks Ms. Hazel Corray for her impeccable typing, attention to detail and cheerful assistance.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER 1

INTRODUCTION

1.1 Problem Statement and Objectives

Computer systems can be viewed from three different aspects:

1) Structure

2) Function

3) Performance

The tasks that the system is expected to accomplish, constitute its function. Structure refers to the organization of the system components, and performance is a measure of how well the system accomplishes its function. Thus a well-designed computer system is one whose structure is such that it accomplishes its function to meet some performance and cost constraints. It is thus very important to understand the relationship between the performance of a computer system and its structure and function. This is true both during the design phase of a new system, as well as in the re-configuration of an existing system to optimize its effectiveness.

In this dissertation, we will present a hierarchical approach to the performance evaluation of computer systems. We will show that a hierarchy of system performance models is a cost-effective way of examining computer system design questions. We will lay down the characteristics that such a hierarchy should possess. We will then develop a procedure for the use of such a hierarchy in the design of a computer system. The principles embodied in our approach will be exemplified by a case-study of the design of a computer system.

In earlier work [KUM76a, 76b], we developed a technique for modelling a system, called control stream modelling. This is an effective

technique for the performance evaluation of complex computer systems by simulation. In this dissertation, we will use such a control stream model in our case-study.

1.2 Background

Performance evaluation of proposed computer systems, for the purpose of examining design questions, is not a new concept. For example, Ballance et al. [BAL62] describe a simulation model that was used in the design of the look-ahead unit of the IBM Stretch system. Boland et al. [BOL67] discuss a simulation model used in designing the memory unit of the IBM System 360/Model 91. However, these were used only to examine a few very specific system design questions. We know of no work in the field that looks at major overall architectural design questions, or a cost-effective tool for examining them. It is our belief that interrelationships between system design parameters can be understood, and real tradeoffs made, only when global, many-parameter models of the system are constructed and analyzed.

Hierarchical approaches to modelling have also been examined in the past, [SEK72, BRO72, BHA76]. However, these have been concerned with the reduction in the complexity of analytic models, by structural decomposition of the system model to form sub-system models that can be analyzed independently. Thus all the models in the hierarchy use the same modelling tools. We believe that ours is the first attempt to bring together a variety of state-of-the-art modelling tools, whose intrinsic range of cost and complexity make them very suitable for use in a hierarchy. We also believe that analysts in the past, have not laid enough emphasis on proper calibration and validation techniques for models. This dissertation will deal in detail with such concerns.

1.3 Structure of the Dissertation

In Chapter 2, we introduce some basic performance evaluation and modelling concepts. We then discuss the motivation behind the hierarchical approach to system modelling. The characteristics and construction of such a hierarchy are then described. Finally, we provide an overview of a system optimization procedure that uses a hierarchy of system performance models.

Chapter 3 introduces the system chosen for the case-study - the CPU-memory subsystem of the IBM 360/91. After a short description of the system architecture, the hierarchy of models used to analyze its performance is described. The two models used are a control stream model, alluded to earlier, and an analytical model built by regression techniques. The models are described in some detail in this chapter.

In Chapter 4, we develop a procedure for optimizing a system design, with respect to some objective function that includes system performance as a component. Using the hierarchy in the procedure, ensures accuracy of performance predictions, and convergence to an optimum system, and at the same time renders the procedure cost-effective. We also attempt to bound the approximation error of the hierarchy, and to estimate the efficiency of the optimization procedure as compared with some simple benchmark procedures.

Chapter 5 discusses the results of applying the procedure to the system chosen as a case-study, for three program traces. The procedure is shown to converge, if not to the exact optimum system, at least to within a near region of the optimum. Sensitivity analysis then identifies the exact optimum besides determining the sensitivity of the objective function to system paramter changes near the optimum.

Chapter 6 summarizes the research and offers suggestions for further research.

CHAPTER 2

A HIERARCHICAL APPROACH TO PERFORMANCE EVALUATION

2.1 Introduction

In this chapter, we first introduce some basic performance evaluation and system modelling concepts. We then attempt to classify models with respect to a variety of features. The hierarchical approach to performance evaluation is introduced and the construction and characteristics of such a hierarchy are discussed. Justification of a hierarchy of models as a powerful tool for the cost-effective design of computer systems is then given and an overview of a procedure that uses a hierarchy to optimize the design of a computer system is presented.

2.2 *Performance Evaluation Concepts*

The performance of a computer system is defined as the effectiveness with which the system handles a specific application. Various measures can be used to describe the performance of a computer system, one of the most common being the system throughput, i.e., the number of tasks processed by it in unit time. Once a measure has been chosen as the one that describes the system performance most satisfactorily, performance evaluation can be viewed from two different aspects:

1) The determination of the performance function F, such that

$$\text{System performance} = F(av_1, \ldots av_m, wv_1, \ldots wv_n)$$

where the av_i are the system architecture parameters, and the wv_j are the system workload parameters.

2) The estimation of values of the above performance function for a specific set of system parameter values

$$(av_1, \ldots av_m, wv_1, \ldots wv_n).$$

2.3 System Modelling Concepts

Any analysis of a system is only an analysis of a model of the system. This is true of system performance evaluation, which is an analysis of one aspect of the system - its performance. A model of a system can be defined as an abstraction that contains only the significant variables and relations of the system. We now discuss a few aspects of system modelling.

2.3.1 Types of Models

Models used for system performance evaluation can be divided into three broad classes [SV076]:

a) Structural Models describe aspects of individual system components and their interactions. They usually serve as the basis for more abstract models, by providing an interface between the real system and the more abstract models. An example is a block diagram model of a system in which each block is a system component.

b) Functional Models define the operation of the system such that the model can be analyzed mathematically or studied empirically. Examples include queueing models that have mathematical solutions for the performance measures of interest, and simulation models that provide empirical evaluations of performance measures.

c) Analytical Performance Models formulate the dependence of performance on the system workload and architectural variables. Such models are usually functions that are fitted to data obtained from functional models.

Some models fall across the above classes as we will see in the next chapter.

2.3.2 Validity of Models

A model is said to be valid when the performance measure values generated by it agree with the actual observations of system performance to within a desired range of accuracy. The range of validity of a model is the region in the multi-dimensional space of system parameters, over which the model is valid. Usually the range of validity and the accuracy of a model have to be traded off.

There are varying degrees of rigor to the validity of models [ZEI75]. At the least rigorous level, a model is replicatively valid, if it matches the performance values already acquired from the real system. At a more rigorous level, a model is predictively valid, when its predictions of performance are corroborated by observations of the system. At the most rigorous level, a model is structurally valid if it not only reproduces the observed system behavior, but truly reflects the way in which the real system operates to produce this behavior.

The choice of the rigor with which a model is judged for validity, depends on the specific purpose that it is being used for.

2.3.3 Other Characteristics of Models

Besides validity, some other aspects of models that we will be interested in, are:

1) Cost: This is usually tied to the computational complexity of the model, i.e., the work involved in using the model to make a single evaluation of system performance. Thus simulation models are usually quite expensive in their computational demands, while mathematical models such as queueing models and analytical performance models are quite inexpensive.

2) Amount of information obtainable from the model: Very often, the analyst is interested in more than a single measure of system performance. For example, in a system which is an interconnection of resources, resource utilization is as important a measure as system throughput, since it can point to system bottlenecks. Models with higher structural validity tend to be capable of yielding more information than models of merely predictive validity. Further, the detail with which such information is available varies considerably. Thus, a queueing model may attempt to yield accurately only the average utilization of a resource, while a simulation model can yield an entire histogram of resource utilization.

2.3.4 Overview of the Model Building Process

Regardless of the type of model chosen, there are certain common features in the process of building up the model as a tool for performance evaluation. The following are some of the basic phases of the model building process:

a) Choice of experimental frame: The experimental frame characterizes a limited region of the entire system parameter space, in which the system is to be modelled. All the aforementioned characteristics of a model are only with respect to the experimental frame for which the model is constructed. Thus a model may be invalid in an experimental frame other than the one chosen, but only its validity in the chosen frame is of importance.

b) Model calibration: Calibration is the process of estimating the parameters that describe the model in the experimental frame. For example, the parameters of an analytical model that expresses performance as a linear function of the system parameters,

$$P = \beta_o + \sum_{i=1}^{m} \beta_i \cdot av_i + \sum_{j=1}^{n} \gamma_j \cdot wv_j$$

are the coefficients β_i ($i = 0$ through m) and γ_j ($j = 1$ through n). The calibration of such a model may involve the fitting of a linear regression equation to observed values of system performance for varying values of the system parameters.

c) Model validation: Once a model has been calibrated, it can be used to predict system performance. Validation is the process of establishing the validity of the model by comparing model predictions of performance with observations of system performance. If the validity is satisfactory, the model predictions in the experimental frame will be accepted. If the validity is poor, the model may have to be recalibrated with the new observations of performance. Calibration and validition for any model can simultaneously improve to a point; beyond that point, they may have to be traded off. Thus, calibration using data from a larger number of observations than the order of complexity of the model may cause poor overall validity in the region. On the other hand, the same model may yield an acceptable degree of validity for more local sub-regions.

d) Prediction using the model: Once a model has been calibrated and validated in an experimental frame, it can be used to predict system performance in that frame. However, if the experimental frame should ever change, the process of calibration and validation will have to be repeated for the new frame.

2.4 A Hierarchy of System Models

We now introduce the concept of a hierarchy of system models for performance evaluation and discuss the motivation behind the concept and

the characteristics that a hierarchy needs to satisfy to be a cost-effective design tool. We also discuss procedures for the construction of a hierarchy.

2.4.1 Motivation Behind the Hierarchy Concept

We will assume that a computer system analyst is evaluating the performance of a computer system for one of the following reasons:

a) To design a computer system which is the optimum system for some objective function that includes system performance as a component.

b) To optimize an existing computer system for an objective function as in (a).

In either event, the analyst is interested in obtaining an optimum system configuration.

Since optimization procedures usually use an iterative scheme to converge to the optimum, a number of evaluations of the objective function will be called for. This requires that the performance component of the function be evaluated with minimum cost, so as to keep the cost of the optimization procedure within reasonable bounds. On the other hand, the performance evaluation must be sufficiently accurate to meet the accuracy demanded of the optimization procedure. A performance model hierarchy provides a cost-effective trade-off between accuracy and computational cost, in much the same way that a memory hierarchy is a cost-effective tradeoff between memory access time and cost.

Further, performance information of varying levels of detail is needed at different stages of the system design process. Thus at the initial stages of design, crudely derived performance information can be used. Later, as the major design features are closer to convergence, more detailed

and accurate performance information will be needed. A performance model hierarchy, as defined below, is compatible with this need.

2.4.2 Characteristics of the Hierarchy

The hierarchy of performance models will have the following characteristics:

The low end of the hierarchy will contain models of high structural, and consequently high predictive, validity. These models tend to resemble the resource configuration of the system. It is expected that they will have a broad range of validity in the system parameter space and that they are capable of yielding detailed performance information of great accuracy. The price to be paid for these desirable qualities is in the high computational demands of these models.

The high end of the hierarchy will contain models of only predictive validity and their range of validity in the system parameter space is much more limited. The performance information that they yield generally has less accuracy than the low level models and is apt to be of a summary, i.e., less detailed, nature. However, they have the advantage of being very much less demanding in their computational requirements.

Intermediate levels of the hierarchy will have intermediate values of these characteristics. Thus travelling up the hierarchy, one sees models that have:

1) Less structural validity
2) More limited range of validity
3) Less detailed performance information
4) Less accurate information
5) Lower computational requirements.

In terms of the types of models described in Sec. 2.3.1, there will be structural models at low levels, functional models at intermediate levels and analytical models at high levels of the hierarchy.

2.4.3 Construction of the Hierarchy

The actual models themselves must be chosen from the state-of-the-art modelling tools available. Thus a typical 3-level performance hierarchy may include a simulation model at the low level, a queueing model at the intermediate level and an analytical model at the high level.

At each level, model calibration will be done using only the performance information of models lower in the hierarchy. Since the model information content increases as we go down the hierarchy, the information obtained from lower level models should be sufficient to calibrate higher level models. Furthermore, calibration may cause a degradation in accuracy. Thus to achieve a certain degree of accuracy for a model, one must use models of higher accuracy to calibrate it. Using only lower level models for calibration ensures this, since accuracy increases as we go down the hierarchy. The calibration procedures will obviously be tailored to the models involved in the calibration.

When the hierarchy is being used to optimize an existing system, the hierarchy can be constructed in a bottom-up fashion since the structural information needed to construct the lower level models is available. However, when the hierarchy is to be used in the design of a system, this information is not available and the construction may have to start at intermediate levels of the hierarchy. For example, an analysis of queueing models of various server configurations can be used in the initial stages to decide the gross structure of the system. As the structure begins to emerge, low-level models can be constructed to examine finer structural detail.

2.5 System Optimization Using the Performance Model Hierarchy

We now outline a procedure for cost-effective use of the performance hierarchy in system optimization. We assume that the system is to be optimized with respect to some objective function that has system performance as a component.

Figure 2.1 is a flow chart depicting the optimization procedure. The *salient features of the procedure are:*

1) The cost of the procedure is kept down by using a high-level model for performance prediction in the iterative procedure that searches for the local optimum.

2) However, the accuracy of the procedure is ensured by the validation check on the high-level model after each prediction of the local optimum. If the check fails, the more accurate low-level model is called at the newly predicted optimum, and the extra information is used to re-calibrate the high-level model in an experimental frame around the new point.

3) After the validation check has proved successful, an analysis is conducted to determine the sensitivity of the objective function to changes in the system parameters around the optimum. This is needed for two reasons:

   a) To locate the true optimum in the local region, since the predictions of the high-level model are accurate only to a certain degree.

   b) To establish the relative importance of the various parameters around the optimum.

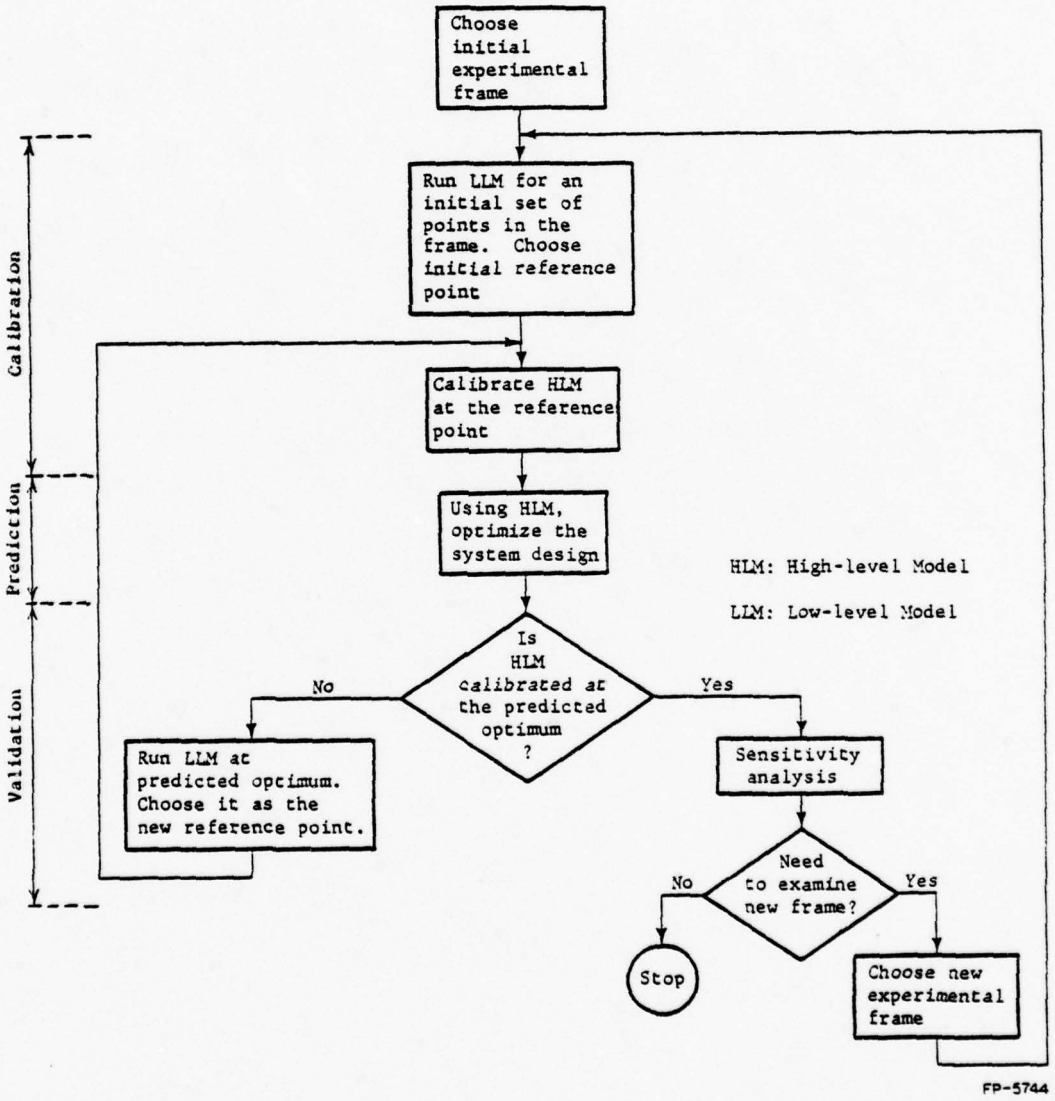


Figure 2.1 System optimization procedure.

Too high a sensitivity to some parameter, may lead the analyst to decide to explore another experimental frame.

In Chapter 4, we discuss such a procedure tailored to study a specific system in detail, touching on aspects such as efficiency and error bounds of the procedure.

CHAPTER 3

MODELS FOR A COMPLEX COMPUTER SYSTEM: THE IBM 360/91

3.1 Introduction

As a case-study of computer system design using the hierarchical approach to system performance evaluation, we chose the CPU-memory subsystem of the IBM System 360/Model 91 as a base. A hierarchy of models was built to evaluate its performance, as a function of some chosen system parameters. The hierarchy was then used to arrive at a system design, in terms of the chosen parameters, that had the optimum cost/performance value. In this chapter, we describe the system and the models in the hierarchy.

3.2 Description of the 360/91

The stated objective of the Model 91 was to attain a performance greater by one or two orders of magnitude over the IBM 7090 [AND67a]. Since circuit and hardware technology advances could provide only a fourfold performance increase, architectural advances were expected to provide the rest of the performance improvement.

3.2.1 Pipelining and Parallelism in the 360/91

The outstanding feature of the Model 91 was the extensive use of pipelining and parallelism throughout the system.

Pipelining is the technique by which the hardware along a processing path is split up into a number of segments with temporary storage between them. Then, when an instruction proceeds from one segment to the next, a succeeding instruction is allowed to use the first segment, even though the first instruction has only barely begun to be processed. Thus, the processing rate is determined, not by the time to traverse the entire processing path, but by the time spent in each segment (see Figure 3.1). The instruction

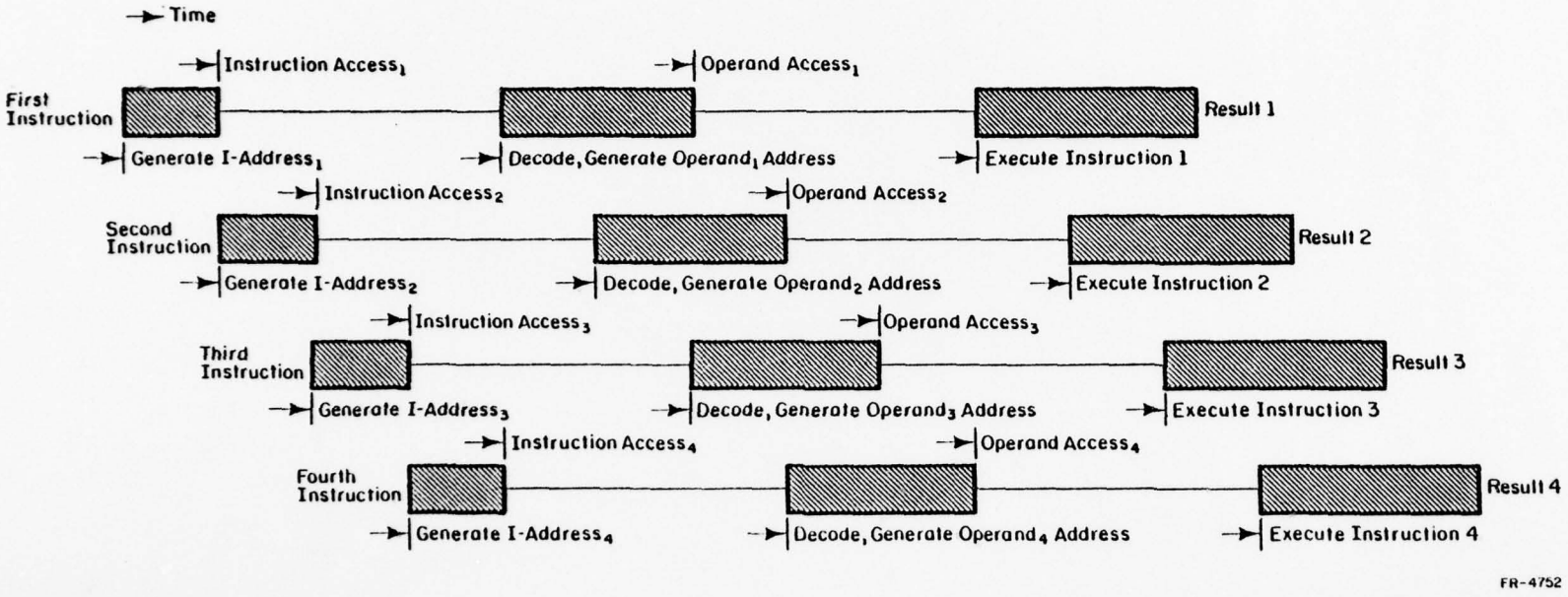


Figure 3.1 Illustration of pipelining between successive instructions.

processing functions of the 91 were split into the segments shown in Figure 3.2. Thus it is possible to enter instructions into this pipeline once every clock cycle, this cycle being 60 nsec.

Parallelism involves the replication of often used hardware units, as well as the possibility of simultaneous use of dissimilar hardware units by different instructions. In the Model 91, the execution function is divided between two separate units - for fixed and floating point instructions, respectively. Further the floating point unit has two separate sub-units - an add unit and a multiply/divide unit. Thus, instructions of these three classes can be executed in parallel.

3.2.2 CPU-Memory Architecture of the Model 91

The organization of the system (see Figure 3.3) will be described under the following division of functions:

1) The instruction unit

2) The execution units

3) The memory unit

4) Buffering in the CPU.

3.2.2.1 The instruction unit

Instructions are pre-fetched from memory and stored in a 64 byte instruction buffer. Pre-fetching buffers the instruction unit against unpredictable memory delays. The instruction unit extracts instructions from the instruction buffer at the rate of one every clock cycle. In the next cycle, the instruction is decoded by the decoder. If it is a fixed or a floating point instruction, it is dispatched to the appropriate execution unit on the next cycle. Concurrently with the dispatching of the decoded instruction, if the instruction needs an operand from memory

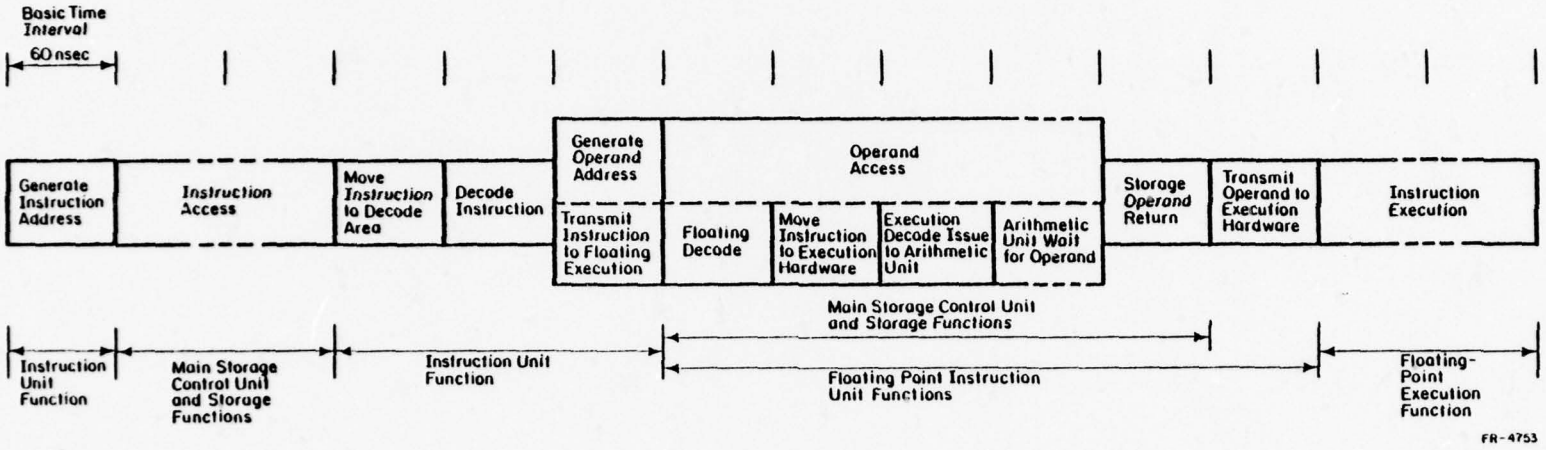


Figure 3.2 Stages in the execution of a typical floating point storage-to-register instruction.

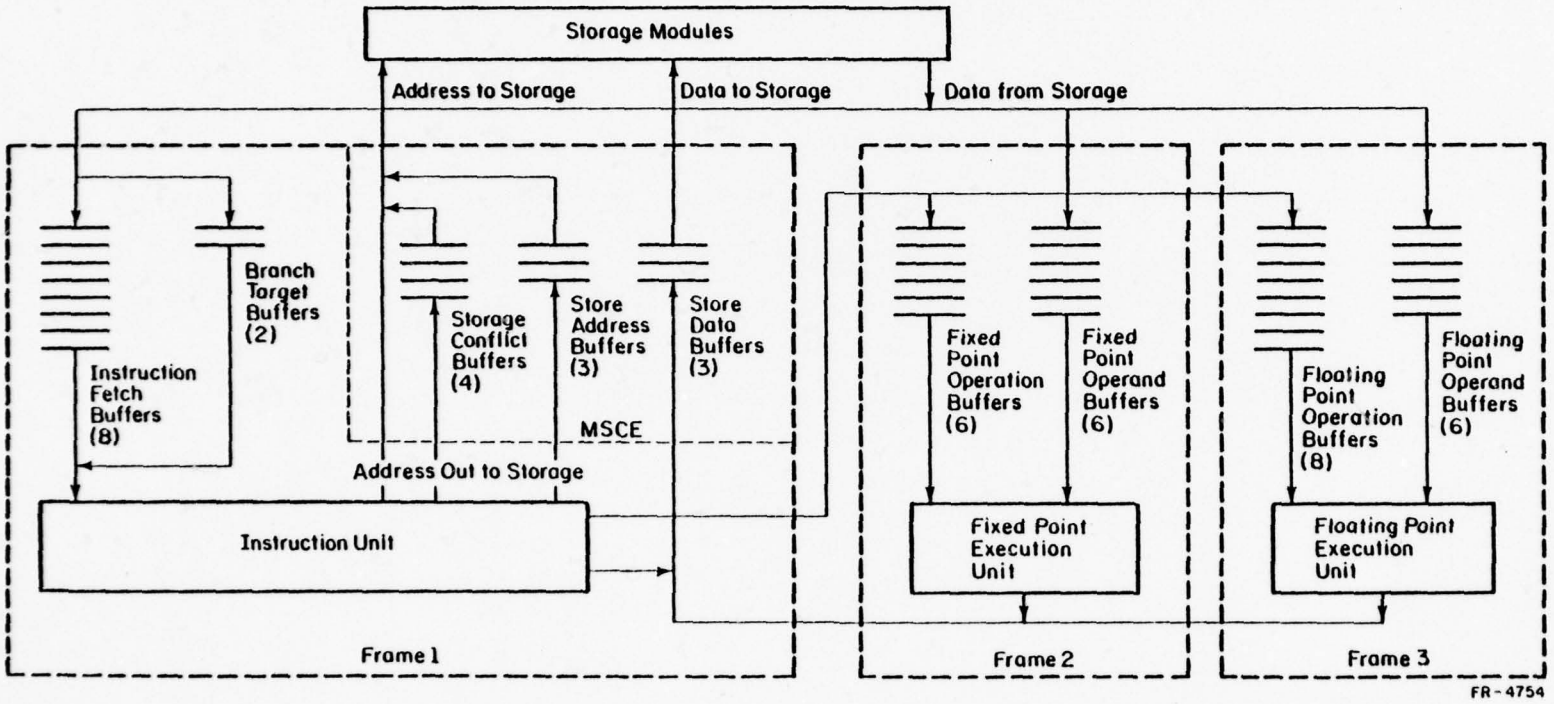


Figure 3.3 System 360/91 organization.

the address parameters (index register, base register and displacement) are combined to compute the operand address, which is then sent to memory on the next cycle, as a fetch request.

The instruction unit also executes branch instructions after decoding them. Unconditional branches cause a switch in the instruction stream, and instruction fetching is done from the target of the branch. If a conditional branch is encountered, and the data on which the branch decision depends has not yet been computed, the CPU enters "conditional mode". Decoding and issuing of instructions continue along the path that reflects the best guess as to the decision of the branch. When the branch is finally decided, these instructions will be cancelled if the guess was wrong, and activated for execution if the guess was right. For most conditional branches, the guess is that it will not be taken. If, however, the target of the branch is back along the stream within 64 bytes of the location of the branch, i.e., for program loops that fit in the instruction buffer, the CPU enters"loop-mode" and assumes that the branch will be taken. For further iterations of this program loop, the instructions will be held in the instruction buffer, and need not be fetched from memory. To further reduce the performance degradation due to a wrong guess, 16 bytes of instruction words are fetched from the alternate branch path and stored in a separate buffer.

3.2.2.2 The execution units

As described earlier, concurrency of execution is increased by having separate units for executing fixed and floating point instructions.

a) The fixed point unit: Within the fixed point unit, execution proceeds serially, one instruction at a time. Many of the instructions require only one clock cycle of execution time.

b) The floating point unit (see Figure 3.4) is subdivided, for further concurrency, into an add unit and a multiply/divide unit [AND67b]. The add unit is pipelined to start an add operation every cycle, and requires two cycles to complete an operation. The multiply unit uses a carry-save adder tree to perform a multiply in three cycles, and an iterative Newton-Raphson technique to perform a divide in 12 cycles. An internal bus, the Common Data Bus, links these units, using the Tomasulo algorithm [TOM67]. It correctly sequences dependent streams of instructions, but permits those which are independent to be executed out of order.

3.2.2.3 The memory unit

Core memory with an access time of 600 nsec and a cycle time of 720 nsec was used in the 360/91. As the CPU clock cycle is 60 nsec, there is a wide disparity between the memory bandwidth and the projected CPU bandwidth. To increase the effective bandwidth of the memory unit, a number of features were incorporated:

a) The memory was 16-way interleaved, i.e., it was split into 16 separate modules or banks, with addresses interleaved so that consecutive addresses reside in consecutively numbered banks. Each bank can be cycled independently, so that, at any given time, more than one bank may be performing an access.

b) Incoming references from the CPU are buffered, if they cannot be honored for any reason such as a bank conflict, unavailability of the data to be stored, or data dependency on a previous, uncompleted reference. Thus requests can be sent to the memory unit at the rate of one per CPU cycle.

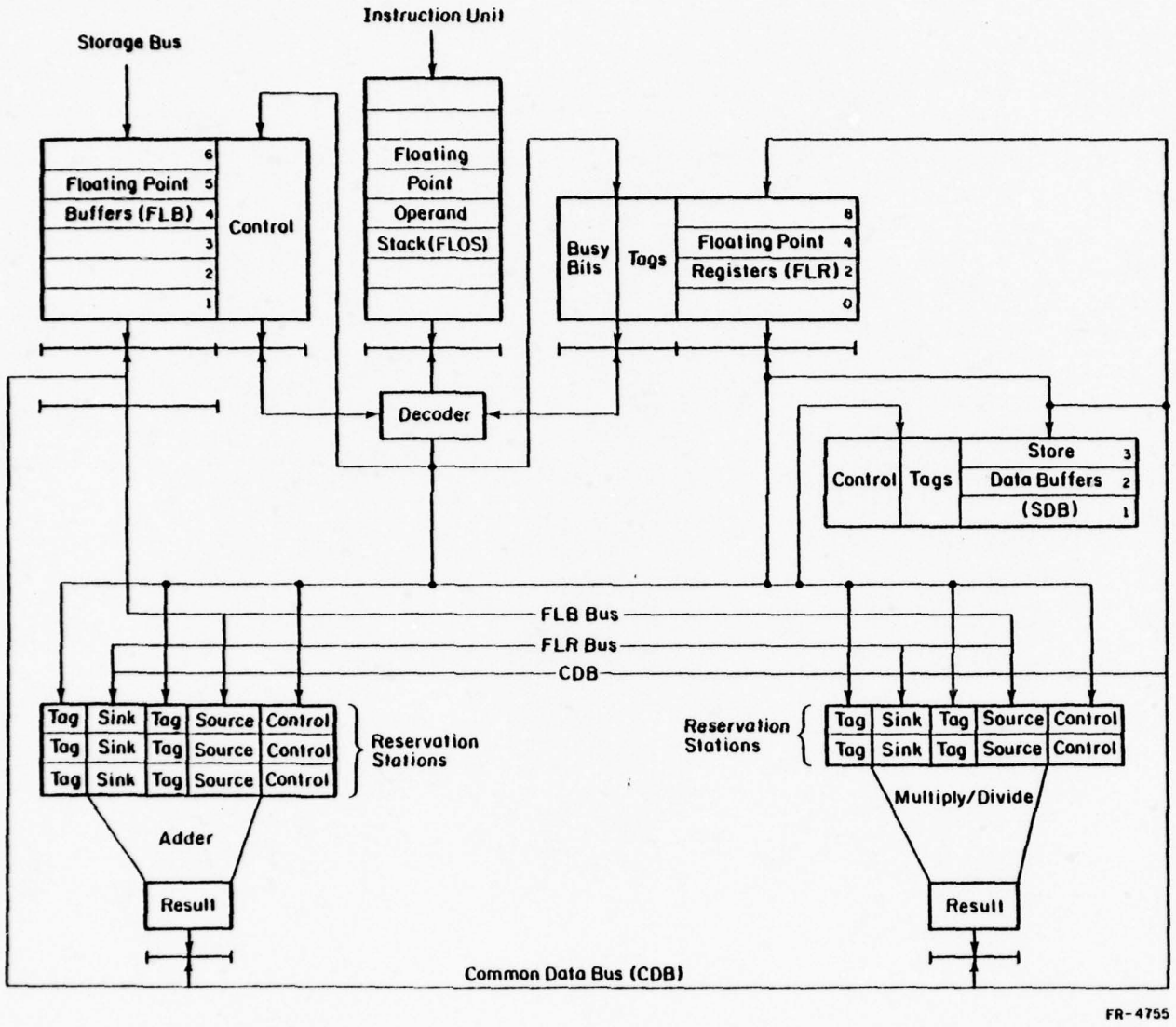


Figure 3.4 The floating point unit organization.

c) A check is made to see if an incoming fetch reference refers to the same location as a previous reference that is currently being serviced. If a match occurs, the second reference can be honored almost at the same time that the data for the first reference is finally available. This is called the multi-access feature [BOL67].

3.2.2.4 Buffering in the CPU

Buffers in the system (see Figure 3.3) provide queueing which smooths the instruction flow. They allow initial segments of the pipeline to proceed with processing despite unpredictable delays down the line due to busy resources, data dependencies or memory accesses. As described earlier, the instruction buffer holds pre-fetched instructions for decoding, and also holds small program loops in loop-mode. In the memory unit, buffers are provided to hold references delayed for any of a number of reasons. Branch target buffers hold instructions fetched from the alternate path of a two-way conditional branch that has not yet been decided.

In the execution units, operation buffers hold decoded instructions sent to them by the instruction unit. Operand fetch buffers provide storage into which the memory returns operands, to be used by the execution units when necessary. Operand store buffers hold operands sent by the CPU until they are stored in memory.

Thus buffering plays an important role in ensuring autonomous execution in the various functional units.

3.3 Overview of the Model Hierarchy for the System

We now outline the hierarchy of models used in the performance analysis of the case-study system. The hierarchy consist of two levels:

a) A control stream model at the low level

b) An analytical model at the high level.

The control stream model is a simulation model, that is driven by a control stream derived from program traces. It is a hybrid between a structural and a functional model (see Sec. 2.3.1), in that its resource configuration, while resembling that of the real system, is an approximation of it. As such it has reasonably high structural validity, yields accurate and detailed information, but is computationally demanding. It should be pointed out that large studies of this kind should use a model of even greater structural validity at the lowest level, i.e., without some of the approximations that were incorporated in the control stream model used in this research. However, for this study, the control stream model is taken as the structural model for the system.

The analytical model is a linear, first-order regression equation, linking system performance, i.e., instruction throughput, with the system parameters. It is predictively valid only in the limited region of its calibration, yields values of only one performance measure, but is trivial to compute.

3.4 A Control Stream Model of the System

In this section, we describe a control stream model of the IBM 360/91 CPU-memory system. This model is the low level in the two level hierarchy of models used to study the system. Consequently its complexity and computation time are significant. To predict system performance using the model, a simulator of the model has to be built, and driven by control streams representative of programs in a desired environment. The model can be used to provide a wide variety of performance statistics of the system.

The model described in this section is a simplification of an earlier control stream model of the system, that is described in [KUM76a] and [KUM76b]. The assumptions about the system that are reflected in this model, are explained in Sec. 3.4.9.

3.4.1 The Control Stream Concept

The simulator of a control stream model is not intended to perform any real computation. The sole purpose of the model is to provide timing and resource usage statistics for typical system usage. Recognition of this fact enables a significant reduction in model complexity, by the introduction of the concept of a control stream.

In the real system, an instruction, while it is being fetched from memory and processed by the CPU, traverses a flow path in the system. This flow path through the system is different for different types of instructions. Moreover, in concurrent CPU-memory systems, the data that is needed by an instruction will have its own independent flow path through the system. Typically, both the instruction and its data traverse their flow paths simultaneously.

The model of the system consists of resources which correspond in some fashion to the resources comprising the real system. In the model, a unit of traffic, or process, is generated corresponding to the starting of an instruction along its flow path in the real system. However, no distinction is made in the model between the instruction flow and the data flow caused by that instruction in the real system. The two taken together form the control flow of that instruction and are reflected in the flow path of the corresponding process in the model. Thus the instruction and data streams in the real system are replaced by a control stream in the model.

The traffic for a simulator of the model, is derived from a program execution trace by one of the methods to be discussed in Sec. 3.4.3. During simulation, however, no attention is paid to the actual data used or produced by the program. Thus the model will be concerned only with the data flow path (inasmuch as it is a portion of the control flow path) and not with the data itself. Since only timing statistics are important, the processing of a traffic unit by a resource in the model consists solely of occupancy of the resource by the traffic unit for the characteristic period of time for that resource in the real system.

3.4.2 Assignment of Logical Resources in the Model

The model associates a logical resource with combinations of various steps in the execution sequence of an instruction. The processing time of each resource is fixed by the combination of execution steps that it represents. Each of these resources can process only one unit of traffic at a time. Thus, the division of the execution sequence and assignment to associated resources is made only as fine as needed to describe the degree of concurrency possible in the system. For example, if there are two distinct consecutive steps in the execution sequence which can never be simultaneously in progress for two different instructions, and if the output of the first step is the only input to the second step and to no other step, then a single resource in the model is assigned to the combination of the steps.

A consequence of this technique is that the model will have no more resources than required to reflect system timing and dependency accurately. This assignment reduces the model complexity significantly over one which assigns a resource to each logical execution step. For

example, a system with no concurrency, i.e., no instruction look-ahead and no execution unit pipelining or parallelism, is modelled as a single resource with variable, but deterministic, processing time.

3.4.3 Control Stream Generation

To exercise the simulator of the model, a control stream to be processed by the simulator must be generated. Since the simulator does not perform any stream computations, each stream instruction need only be sufficiently described so as to enable the simulator to determine its dynamic flow. This information would minimally consist of:

1) The static control flow path of the instruction, i.e., the resources needed to process the instruction in the order that they are needed. For example, in a concurrent system, some of the resources needed by an instruction may be the instruction decoder, a particular execution unit, a memory location from which an operand is to be fetched and busses to transmit the operands to the execution unit.

2) The dependency of this instruction on instructions preceding it in the stream. This information is necessary for the simulator to set up the interlocks to ensure correct sequencing of the stream. The execution of a program on a concurrent processor gives rise to three kinds of dependencies [TJA70]:

a) Data dependency: this occurs when two instructions reference the same operand location. In a concurrent system, they have to be processed so that they reference that location in the correct sequence as dictated by the program.

b) Procedural dependency: this occurs when there is a conditional branch instruction in the stream. Execution beyond the branch cannot proceed until the branch decision is made and one of two paths is chosen for execution.

c) Operational dependency: this is caused by two instructions attempting to use a processor resource at the same time. This results in a conflict that has to be resolved by some priority mechanism.

Thus, for a control stream to be executed by the simulator of the model of a concurrent system, the data dependency information for an instruction would point to the most recent instructions that read from or wrote into the operand locations referenced by this instruction. The procedural dependency information would point to the most recent conditional branch instruction which must be executed before this instruction is executed. Note that the control stream represents a single execution of a single program. Thus all activity following branches is actually known by the simulator *a priori*. Execution is merely delayed until such time as the branch would have been completed in the real system. The operational dependency of the instruction is completely specified by its static control flow path through the resources of the system.

3.4.3.1 Control stream generation from program traces

The instruction execution trace of a real program is gathered while it executes on the real system that is to be modelled, or a compatible system. Each instruction in the trace is then mapped into a control stream instruction, specified by the set of parameters needed to describe it to the simulator. The static control flow path of each instruction is entirely determined by the types of operands that it uses and the operations

that it performs on them. Data dependency information is gathered from a simple forward scan of the trace by maintaining a list of operands used and the most recent instructions that used them. This list need only keep track of dependencies on a certain number of most recent instructions, on the grounds that instructions further back would have completed execution and will not delay instructions far ahead in the stream. For every instruction, the data dependency information is then derived by scanning the list for the operands used by this instruction and specifying the most recent instructions to use those operands. The list is then updated. Data dependency interlocks built into the simulator use this information to prevent improper out-of-sequence usage of operands. Procedural dependency is specified by the occurrence of conditional branches in the stream, and their "data dependencies" - thus, no extra scanning of the trace is necessary to gather this information. Operational dependency is specified by the static control flow paths of the instructions in the stream - here too, no extra scanning of the trace is necessary.

3.4.3.2 Synthetic control stream generation

To synthesize a control stream, a comprehensive, yet tractable, model of the workload has to be used as the base. One approach to modelling the workload is by statistical means. The workload of programs in a given environment can be characterized by a number of statistical distributions. The information obtainable from these distributions must be sufficient to derive the main attributes of control streams described earlier. For example, resource demands of the control stream can be derived from an instruction frequency distribution. Data dependency information for

instructions in the control stream can be derived from a distribution of the number of intervening instructions between two instructions accessing the same operands.

Procedural dependency arises from the occurrence of algorithmic control constructs in programs. Almost all the branch instructions in programs can be attributed to the occurrence of one of the following high level language features: conditional constructs (if-then-else and case statements), iterative constructs (for and while statements) and procedures (calls and returns). Thus we feel that the procedural dependency information for a control stream is best derived from distributions describing the occurrence of high level language features in that class of programs. For example, iterative constructs can be described by distributions of the iteration count and the length of the iteration (in instructions).

We now outline a procedure for stream generation using these statistical distributions. The instruction frequency distribution is sampled, to decide the resource usage pattern of the next instruction in the stream. If it is an instruction that can have a data dependency, data dependency information is generated for it by sampling the data dependency distributions. If it is a branch instruction, a high level language construct will be generated, depending on the type of branch at hand. For example, a branch-on-counter-condition instruction, such as BXLE or BXH on the IBM 360, is most often used with for-loops by computers, and will trigger the generation of a for-loop construct in this scheme. This will include generation of an iteration count and the length of the iteration (in instructions) from the corresponding distributions. A

procedure similar to the above is then followed for generating the instructions in the construct. When the entire construct has been generated, the outer procedure for generating the main stream is continued. The stream length is chosen by sampling the program length distribution (in instructions), and the generation procedure is stopped when this length has been reached.

The above procedure follows a first order approximation since it assumes that there is no correlation between the occurrence of successive instructions in programs. More refined procedures would replace the instruction frequency distribution by higher order distributions that describe the occurrence of instruction pairs, triplets, etc.

3.4.4 Terminology Used in the Model Description

The simulator of the model was implemented in SIMULA [BIR73]. Consequently, much SIMULA terminology has been used in the description of the model that follows. The basic time unit of the model is one clock cycle of the CPU. The units of traffic flowing through the system are processes - these are the dynamic entities of the simulation. A resource models some consecutive stages in the execution process, as described in Sec. 3.3.2. A buffer has the same function in the model as in the system - temporary storage for a process while it waits for a certain event to occur.

When a process needs a resource, it gets control of the resource, occupies it for the characteristic time of that resource and then relinquishes control of the resource. If the resource is not available, i.e., it is occupied by another process, the process waits, either in a buffer or, in the resource that it is currently occupying, until that resource is freed and this process has the highest priority among those waiting to use that resource. As a consequence, a process frees a resource that it is occupying,

only after it has acquired the resource that it needs next, or after entering a buffer where it will wait for its next resource.

To model distinctive sections of the control flow path of an instruction, different processes are used. Thus an instruction process models the flow through the instruction unit and the execution units. An operand process models the independent flow of an operand that the instruction needs. A memref process models the flow through the memory unit for any memory reference - an instruction fetch, an operand fetch or an operand store. Thus the control flow of a single instruction may involve the creation and termination of many processes, depending on its path. Appropriate synchronization mechanisms have to be provided for communication among all these processes. Further, the entity in the real system that a process models may change with time. Thus an operand fetch from memory involves the following sequence of actions:

a) The instruction process that needs the operand creates an operand process to model the computation and the transfer of the operand address to memory. This operand process may undergo delays due to resource conflicts, data dependency, etc. The instruction process, in the meantime, traverses its control flow path concurrently.

b) The operand process creates a memref process to model the memory access and waits for it to return.

c) The memref process may undergo delays due to memory conflicts etc. On completing the memory access, the memref process signals the parent operand process and is terminated.

d) The operand process now models the actual operand to be transferred to the execution unit. After doing so, it signals the parent instruction process and is terminated.

In the description of the model, a process and the type of entity that it models in the real system will be used interchangeably. For example, "instruction" will be used in place of "instruction process," except when ambiguity may result. Further, in place of the pseudo-processing that a model resource does, the function accomplished by the corresponding resource in the real system is quoted for descriptive purposes. For example, an instruction process will be described as being "decoded in one cycle," whereas all that occurs in a simulation of the model is that the process occupies the resource modeling the decoder in the real system, for one cycle. In a similar manner, an instruction process will be quoted as "fetching an operand from memory" to denote the memory operand fetch sequence described earlier.

3.4.5 Resources and Buffers in the Model

1) IBUF - the instruction buffer: holds pre-fetched instructions.
2) IEX - the instruction extractor resource: extracts the next instruction from IBUF in 1 cycle.
3) IDEC - the instruction decoder resource: decodes the instruction sent to it by IEX in 1 cycle.
4) FXIU - the fixed point unit instruction decoder resource: decodes fixed point instructions in 1 cycle and executes fixed point loads and stores.
5) FXEU - the fixed point execution unit resource: executes fixed point computational instructions. Most of the instructions take 1 cycle, with multiplies and divides executing in 11 and 36 cycles respectively [RIS72].

6) FLIU - the floating point unit instruction decoder resource: decodes floating point unit instructions in 1 cycle and executes floating point loads and stores.

7) FLAD1 and FLAD2 together constitute the floating point unit add resource: Each represents one segment of a 2-segment pipeline that executes floating point add instructions in 2 cycles, but can start a new add operation every cycle.

8) FLMD - the floating point multiply/divide resource: executes floating point multiply and divide instructions in 3 and 12 cycles respectively.

9) The memory bank resources: each holds a memory reference process for a number of cycles equal to the memory cycle time. The number of banks is a model parameter.

Figure 3.5 shows the interconnection of these resources and buffers. The figure reflects the approximations made by the model to be discussed in Sec. 3.4.9. Thus no busses are shown because, even though the nominal bus transfer time of 1 cycle is included in the control flow, the model assumes that there is never any contention for the use of these busses. This is true of the operand address generation resource as well. The path for branch instructions out of IDEC - indicates their termination after execution, while the path for "aborted instructions" out of IBUF, indicates the termination for instructions that were not decoded because they followed a branch that was taken.

3.4.6 Control Flow of an Instruction Process

The instruction process models the control flow of an instruction through the instruction and execution units. We now describe the sequence of events in the life of an instruction process. The description is not

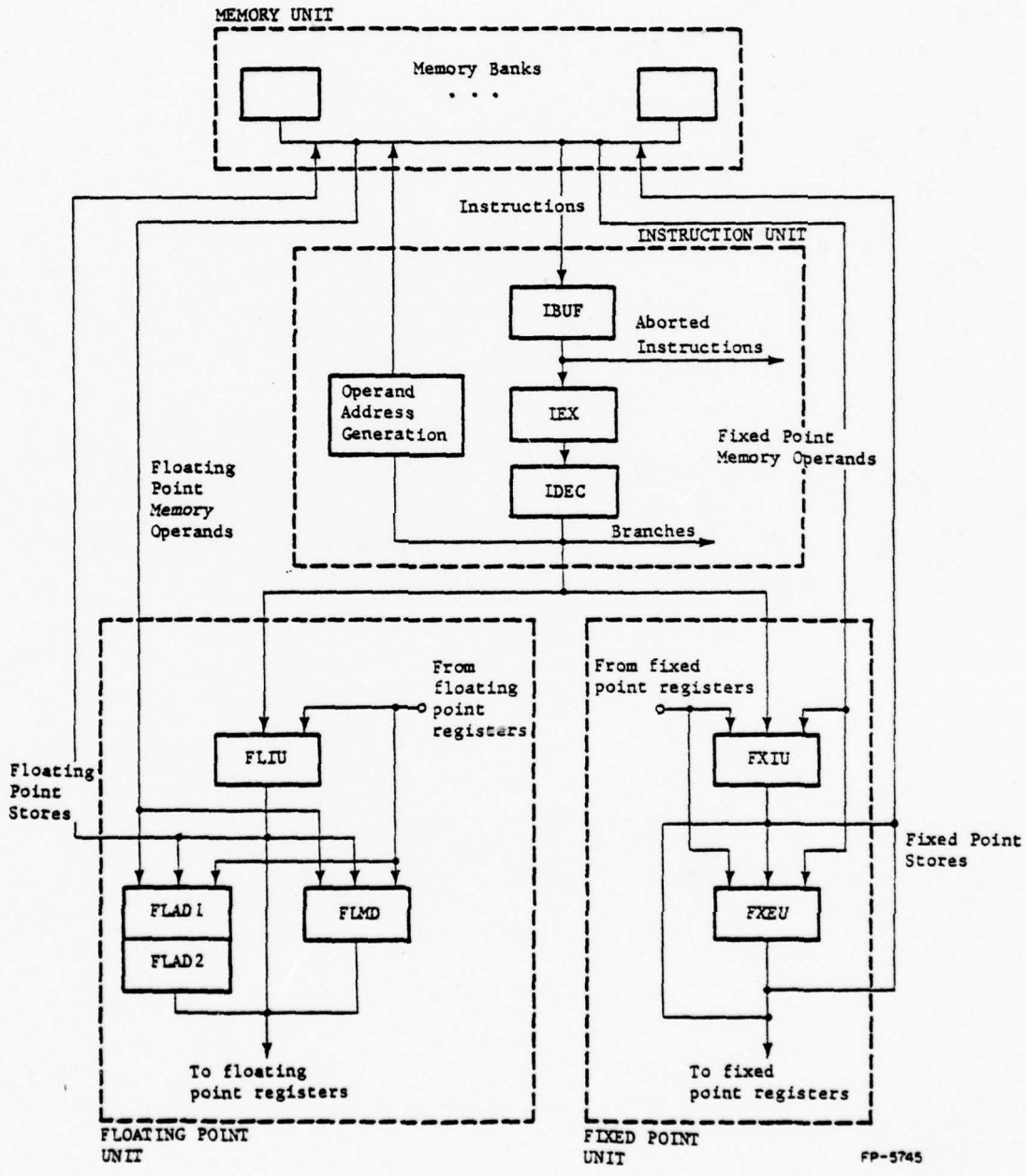


Figure 3.5 Resource configuration in the control stream model.

complete in all respects, for lack of space. Detailed documentation is provided in the listing of the simulator program in Appendix B.

3.4.6.1 The flow common to all instructions

1) a) If the CPU is not in loop-mode, the instruction process starts by making a memory reference to model the instruction fetch, i.e., it invokes a memref process and waits for it to return from memory. When the memref is done, the instruction enters the instruction buffer IBUF in its proper place, i.e., that which maintains the program instruction sequence.

b) If the CPU is in loop-mode, no instruction fetch is necessary and the process starts with the instruction in IBUF itself.

2) When it has reached the head of the queue of instructions in IBUF, the instruction acquires the instruction extractor resource and leaves IBUF. It then schedules a new instruction process to model the prefetch into the vacant slot in IBUF. After 1 cycle in IEX, it acquires the decoder IDEC, releases IEX and is decoded in 1 cycle.

3) At this stage, if any of the address registers needed for an operand address computation by the instruction, is unavailable, i.e., has not yet been updated by a previous, as yet uncompleted instruction, this instruction waits in IDEC, delaying the instruction stream behind it.

4) If the instruction is not a branch, and needs an operand to be fetched from memory, an operand process is created now. This process will model the operand fetch from memory, and will return independently to the execution unit to merge its control flow with that of the parent instruction process.

The above steps are common for all instruction processes. The control flow of the process from this step on depends on the type of instruction that the process models.

3.4.6.2 Branch instructions

a) When the CPU is not in loop-mode:

a.1) If the instruction is an unconditional branch, the instruction stream has to be switched to the branch target. The branch instruction process causes the termination of all the instructions in IBUF and all outstanding instruction fetches (see the "aborted instructions" path in Figure 3.5.). It then initiates new instruction fetches from the target, and is then terminated.

a.2) If the instruction is a conditional branch, whose decision depends on the value of the condition code, or a counting register, there may be a delay before the branch decision is made. Since the system, in this case, assumes that the branch will not be taken, in the model, decoding continues in conditional mode. However, since the branch decision is already known to the simulator, it decodes and issues the actual instru tions following the branch in the control stream only if the branch will not be taken and dummy instruction processes, if it will. Later, when the branch decision is "made," the conditionally forwarded instructions will be activated or the dummy instructions cancelled, respectively. In the latter case, IBUF is also emptied and new fetches from the branch target are initiated. The branch instruction process is then terminated.

In both of the above cases, if the branch is taken and the target is back along the instruction stream, at a distance from the branch location that is smaller than the size of IBUF, the CPU is switched to loop-mode.

b) When the CPU is in loop-mode

b.1) If the instruction is an unconditional branch, the stream has to be switched to the branch target, but no new fetches are necessary.

All that occurs is that the instruction at the target, which is already in IBUF, is scheduled next for decoding. The branch instruction process is then terminated.

b.2) For conditional branches, conditional mode is set until the branch decision is made. However, the policy for conditional decoding of instructions is reversed from that of the non-loop-mode case. The system now assumes that the branch will be taken. Thus, in the model, actual instructions from the target of the branch in IBUF are sent for decoding if the branch will be taken, and dummy instruction processes, if it will not. When the branch decision is "made," the conditionally forwarded instructions will be activated, or the dummy instructions cancelled, respectively. In the latter case, IBUF is emptied, loop-mode is turned off and new fetches from the sequential path following the loop are initiated. The branch instruction process is then terminated.

3.4.6.3 Fixed point instructions

1) After passing through the instruction unit as described in Sec. 3.4.6.1, a fixed point instruction process is transferred in one cycle to the fixed point execution unit. If it is a conditionally issued instruction, it cannot proceed for execution until the conditional branch that set conditional mode has been decided.

2) When the instruction reaches the head of the queue of instructions in this unit, it is ready for execution. Its subsequent control flow depends both on its type and the type of architecture that is being modelled for the fixed point unit. The latter is an overall model parameter and can take one of three values:

a) Serial: In this architecture only one instruction may be in process at a given time in the entire fixed point unit. Thus the instruction gets control of the fixed point decoder FXIU, only after the previous fixed point instruction has completed execution in the unit and transferred its result to the appropriate destination.

b) Pipelined: In this architecture, the decoding in FXIU and the execution in the execution unit FXEU, are pipelined. Thus, an instruction can be decoded in FXIU, while the previous instruction is still using FXEU. If the second instruction is a load or a store, and has its operand available, it can proceed simultaneously and even finish before the first. If its operand is not available, it waits in FXIU until it is, thus delaying subsequent instructions. If the second instruction is not a load or a store, and needs the FXEU to execute, it has to wait in FXIU, until the first has finished execution. Further, when the second instruction needs the result of the first instruction as an operand, it obtains that result from the appropriate location, after the first instruction has transferred it there.

c) Dataflow: This architectural type models the floating point unit architecture of the 360/91 as designed by Tomasulo [TOM67], and the architecture discussed by Dennis [DEN74]. The FXIU, after decoding an instruction, executes it if it is a load or store. If it is neither, the FXIU deposits it in a buffer, that creates the effect of a number of virtual execution units. These are called reservation stations in [TOM67]. The FXIU is now free to decode subsequent instructions.

The virtual units acquire control over FXEU (the real execution unit) in the order in which they become ready for execution, i.e., when they have received all their operands. Thus instructions that do not depend on one another can be executed out of sequence. Further, when an instruction

is completed and has a result to be transferred, it broadcasts the results to all the virtual units that need the operand in the same cycle. This eliminates a number of redundant operand transfers.

3) The instruction, after it gains control of FXIU, is decoded in 1 cycle. If it needs two operands, the instruction process itself models the control flow of one of these - the register operand. If the other is a memory operand, the operand process to model the fetch has already been created (see Sec. 3.3.6.1). If the other is a register operand, or if the instruction needs only a single register operand, the instruction now creates an operand process to model the control flow of that register operand.

If the register operand that this process now models is available, its transfer to the execution unit or to the destination of a load or store instruction, takes one cycle. If any operand is not available, the action taken depends on the type of architecture being modelled, as described earlier.

4) If it is a load, the instruction has now been completed. If it is a store, a memref process is created to model the storing of the operand in memory, at the end of which the instruction has been completed. If it is neither a load nor a store, the process occupies FXEU for the required execution time of the instruction that it is modelling. At the end of its execution it transfers its result to the appropriate destination(s) in one cycle.

The instruction process is then terminated.

3.4.6.4 Floating point instructions

The control flows for floating point instructions are very similar to those for fixed point instructions.

1) A floating point instruction is transferred from the instruction unit to the floating point unit in one cycle. If it has been conditionally

issued, it cannot proceed for execution until the branch that set conditional mode has been decided. When it reaches the head of the queue of instructions in this unit, it is ready to be decoded by the floating point decoder, FLIU.

2) Its subsequent control flow depends on its type and the type of architecture being modelled. The three types of architectures are analogous to the three types of fixed point unit architectures. However, there are two execution units in the floating point unit; the floating point add unit FLAD and the floating point multiply/divide unit FLMD. With this difference, the three types are:

a) Serial: Only one floating point instruction may be in process in the entire unit at any given time. Thus the FLIU can decode the next instruction only after the previous instruction has completed execution, and transferred its results to the appropriate destination.

b) Pipelined: In this architecture, decoding and execution are pipelined. Thus decoding in the FLIU, an add in the FLAD and a multiply or divide in the FLMD can proceed simultaneously. Loads and stores, which are executed in the FLIU itself, may thus finish even before previous add or multiply instructions, if their operands are available. If not, they hold the FLIU, until the operands do become available. Adds and multiplies decoded in FLIU, wait until their respective units are free before releasing the FLIU. Results are transferred to their destinations, from where following instructions can obtain them.

c) Dataflow: Both FLAD and FLMD have their sets of reservation stations, which act as virtual execution units. The FLIU executes loads and stores and deposits other instructions in the appropriate virtual execution units. It can thus decode instructions at the rate of one every cycle. Instructions in virtual execution units acquire the physical units,

in the order in which they become ready for execution. After execution, the result is transferred in one cycle to all the virtual units that need it.

3) After being decoded by the FLIU in one cycle, the instruction obtains its operands and completes execution in the same manner as fixed point instructions described in Sec. 3.3.6.3.

3.4.7 Control Flow of an Operand Process

The operand process models the control flow of operand fetches which proceed in parallel with the control flow of the main instruction. Since fixed point and floating point operand processes have analogous control flows in their respective units, we present a common description of both types.

If the process models a memory operand fetch, it has the memory address computed in one cycle and is transferred to the memory unit in another. In the memory unit, it waits until the most recent instruction that needed that operand, for reading or updating, has used it. When this data dependency has been resolved, it creates a memref process to model the actual fetch of the operand from memory. When the memref process signals this process on return, the operand has been fetched from memory and can be transferred to its destination. The actual transfer depends on the type of architecture being modelled. In serial and pipelined architectures, the operand waits until the instruction that needs it has acquired the resources necessary for its execution. In a dataflow architecture, the operand waits until the instruction that needs it has acquired a virtual execution unit. In all the above cases it is then transferred to its destination, an execution unit or a register, in one cycle.

If the process models a register operand fetch, its control flow again depends on the type of architecture being modelled. In serial and pipelined architectures, it waits until the most recent instruction that needed that operand, for reading or updating, has used it. When this data dependency has been resolved, the operand is transferred to its destination - an execution unit or a register - in one cycle. In a dataflow architecture, if the operand is available for use, it is transferred in one cycle to its destination. If not, it will be transferred automatically to the virtual execution unit, when the most recent instruction that needed to update it has been completed. Thus the operand process need not wait for data dependency to be resolved.

At this stage, for both register and memory operands, the process signals its parent instruction that the operand has been delivered and is terminated.

3.4.8 Control Flow of a Memref Process

The memref process models the control flow of any memory reference through the memory unit. When it is invoked, the address of the memory reference has already been transmitted to the memory unit. The memref process waits until all previous references to the memory bank that is addressed by this process, have been completed. When the bank is available, this process occupies it for a number of cycles equal to the memory access time being modelled. It then signals the parent process that invoked it that the access is complete. After this, it continues to occupy the bank until a number of cycles equal to one memory cycle time have passed since its initiation. It then releases the bank, and is terminated.

3.4.9 Approximations Made in the Model

A number of approximations were made in building the model, in order to keep it tractable. Further, experience gained from the more detailed model in [KUM76a] indicated that a number of features had a negligible effect on the system performance. We now list some of these approximations.

1) Except for the instruction buffer, all the system buffers are assumed to be unbounded in size. This is reflected in the description of the model, where processes are never delayed because of buffer overflow. Evidence from the more detailed model indicates that buffer overflows rarely occur in the system as the buffer sizes in the original system are generally adequate, yet not costly. However, keeping the instruction buffer bounded is the most effective way of keeping the instruction supply rate in the model close enough to that in the system.

2) Conflicts for busses that transfer data are neglected. In effect, the model assumes an unbounded number of copies of all busses. Since bus conflicts play a very small role in performance degradation, this is not a very serious approximation.

3) In the system, each instruction fetch returns a double word (8 bytes) from memory. This double word can contain from one to four instructions. In the model, a separate fetch is needed for every single instruction. This is a very serious approximation, and was made solely to keep some higher level models tractable. We have not examined the effects of this approximation very carefully.

4) In the system, when a conditional branch is decoded, two double words are fetched from the branch target, as a hedge against an incorrect branch prediction. In the model, this feature is absent.

5) In the system, the memory unit checks the address of each incoming fetch request against the addresses of ongoing fetches or stores. If there is a match, the second fetch can be honored almost simultaneously with the previous reference to the same location. This is called the multi-access feature [BOL67]. In the model, this feature is absent. This assumption reflects evidence from the more detailed model [KUM76a] that multi-access occurs fairly infrequently and does not affect performance to a considerable extent.

In view of the above approximations, and the lack of evaluation of their impact on the accuracy of model predictions, it would be more realistic to say that the system being modelled is a system very much like the IBM 360/91.

3.4.10 System and Model Parameters

To study system performance as a function of various system architectural parameters, a number of these system parameters were parameterized in the control stream model as well. The model parameters that can be explicitly specified are:

1) mc - the memory cycle time (in CPU clock cycles). For simplicity, the memory access time is assumed to be 5/6 of the memory cycle time.

2) mb - the number of memory banks, i.e., the depth of memory interleaving.

3) ib - the size of the instruction buffer IBUF.

4) fx - the fixed point unit architecture. The values associated with the three types described in Sec. 3.3.6.3 are:

```
Serial    :1
Pipelined :2
Dataflow  :3
```

5) fℓ - the floating point unit architecture with the same association of values with types as for fx.

6) ℓm - the loop-mode feature. The value assigned to ℓm is:

0 - if no loop-mode capability exists.

1 - if the loop-mode capability exists.

It should be noted that the bandwidth of each of the major units - the instruction unit, the memory unit, the fixed point unit and the floating point unit - is affected by at least one parameter in the above set. Further, the values associated with the execution unit architecture types, increase in the expected direction of increase of bandwidth.

Besides these, a number of other architectural parameters can be varied by simple changes to the simulator program. These include execution times of various resources, priority mechanisms for scheduling various resources, etc.

3.4.11 Performance Measurements Using the Model

As discussed in the previous sections, the model is used in the construction of a simulator, which is driven by program execution traces. A wide variety of performance statistics can be gathered during the simulation.

For example, suppose this model is to be used to calibrate a queueing network model of the system. From a knowledge of what the servers comprising the queueing model accomplish, different points in the control flow paths of the processes of the control stream model can be identified as the points of entry and departure of these servers. At these points, statistics can be gathered regarding server arrival and departure rates and counts. These statistics can be used to calibrate the queueing model.

If an analytical model is to be built relating some overall performance measure such as system throughput or memory utilization, to the system parameters, the appropriate performance statistics can easily be gathered from simulations of the control stream model, for the required settings of the system parameters.

The performance measure that was most frequently used in the study, is the average system instruction throughput. This is defined as the average number of instructions that were completed per CPU cycle, over the run of the program. This was approximated in the model, by the average number of instruction processes that terminated normally per simulator cycle. Normal termination excludes those instruction processes that were flushed from the system following a branch, as well as those dummy instruction processes that were conditionally decoded following a wrongly predicted conditional branch. This throughput is measured very simply in the model by dividing the total number of instruction processes that terminated normally, by the number of simulator cycles needed to execute the program trace.

3.5 An Analytical Performance Model of the System

In this section we will describe an analytical performance model of the system chosen for study. The model attempts to describe the performance measure of greatest interest - system instruction throughput - as a function of the system parameters listed in Sec. 3.4.10. Statistical regression techniques are used to estimate this function. For an excellent introduction to regression theory, see [DRA66]. [TSA72] is an illuminating example of the application of regression modelling to computer system performance evaluation.

3.5.1 Introduction to Regression Theory

Analytical model building by regression analysis involves an iterative search for a mathematical expression relating a dependent variable Y, called the response, to a set of independent variables X_1, X_2, ... X_n, called the factors, on the basis of observed data. Since the functional relationship may, in general, be complex, a summarization of the relationship is achieved by:

a) Selecting a small but relevant subset of the independent variables for inclusion in the expression.

b) Choosing a simple but plausible mathematical form to express the relationship. A common form to choose is the linear form, in which the expression is linear in its parameters. A further simplification is the first-order model, in which the highest power of any factor that occurs in the expression is one.

3.5.1.1 Linear regression models

In theory, the entire set of observed data can be fitted exactly by a linear model of the appropriate order. For example, a linear quadratic model involving three factors is

$$Y = \beta_0 + \sum_{i=1}^{3} \beta_i X_i + \sum_{i=1}^{3} \beta_{ii} X_i^2 + \sum_{i=1}^{3} \sum_{j=i+1}^{3} \beta_{ij} X_i X_j$$

This model, which has 10 parameters, can be used to exactly fit data from less than 11 observations. However, such a model may have very poor validity at points other than those at which the observations have been made. Further the interpretation of higher order interactions is usually rather difficult. In practice, therefore, it is preferable to start with a linear first-order model and to progressively introduce higher order interactions only if they

greatly increase the precision of the model. A good example of this approach is found in [TSA72]. A linear first-order model of the above system is

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3$$

3.5.1.2 Calibration of regression models

Calibration of say, a linear first-order model consists of estimation of the β_i's, from data observed about the system. The estimation procedure is that of least square error. Suppose we have n observations of a system response Y to which a linear first-order model involving one factor X_1 is to be fitted. Let the observations be $(X_1^1, Y^1), \ldots (X_1^n, Y^n)$. The response value predicted by the model at the factor value X_1^i is

$$\hat{Y}^i = \beta_0 + \beta_1 X_1^i$$

Thus the sum of the squares of the deviations of the model predicted responses from the system responses is

$$S = \sum_{i=1}^{n} (Y^i - \beta_0 - \beta_1 X_1^i)^2$$

For least square error, β_0 and β_1 are assigned values which minimize S. This is done by solving the equations

$$\frac{\partial S}{\partial \beta_0} = -2 \sum_{i=1}^{n} (Y^i - \beta_0 - \beta_1 X_1^i) = 0$$

and

$$\frac{\partial S}{\partial \beta_1} = -2 \sum_{i=1}^{n} X_1^i (Y^i - \beta_0 - \beta_1 X_1^i) = 0$$

which yields the least square estimates of β_0 and β_1.

The above procedure is easily generalized to models of higher order as well as to non-linear models.

3.5.1.3 Regression algorithms

The most widely used regression model building algorithm is the stepwise regression procedure described in [DRA66]. In this procedure, factors are introduced into the regression equation in the decreasing order of their partial correlation with the response. However, a factor is introduced only if its correlation with the response is above a significance level that is specified by the model builder. Moreover, when a factor is introduced, the contributions of factors that had been introduced before are re-assessed, and some of these may now be rejected from the model. Thus the final model will contain only those factors whose correlation with the response is above the significance level specified by the model builder, and which are not strongly correlated with each other.

The detailed procedure involves analysis of variance and other statistical techniques, which are beyond the scope of this report. [DRA66] is an excellent reference for these methods.

3.5.2 The Analytical Model of the System

The analytical model of the system 360/91, that was chosen as the high level of the 2-level hierarchy, expresses the system instruction throughput, as defined in Sec. 3.4.11, as a function of the six system parameters described in Sec. 3.4.10. The model chosen was a linear, first-order model. However, to achieve a better fit with the low-level model predictions, as well as to reflect the actual range of values usually chosen for some system parameters, these parameters were represented in various functional forms in the analytical expression. Thus, since most systems are designed with the number of memory banks and instruction buffer

slots chosen as powers of 2, these parameters were represented in the logarithmic form in the analytical model.

The analytical model is thus expressed by the relation:

$$sit = \beta_0 + \beta_1 \cdot mc + \beta_2 \cdot \log_2 mb + \beta_3 \cdot \log_2 ib + \beta_4 \cdot fx + \beta_5 \cdot f\ell \qquad (1)$$

where:

sit = average system instruction throughput

β_i = model parameters that are estimated by the regression procedure.

mc = memory cycle time

mb = number of memory banks

ib = instruction buffer size

fx = the fixed point unit architecture parameter

$f\ell$ = the floating point unit architecture parameter.

The ℓm parameter is handled by building separate equations as in (1) for $\ell m = 0$ and $\ell m = 1$.

As is characteristic of models at the high end of the hierarchy the accuracy of this model is expected to be good only in restricted regions of the system parameter space. However, its evaluation is trivial - once the β_i's are known, i.e., once the model is calibrated, the performance prediction for a set of system parameter values is obtained by plugging these values into equation (1).

3.5.2.1 Model calibration

Given a set of n observations of throughput, sit_j, at the system parameter setting (mc_j, mb_j, ib_j, fx_j, $f\ell_j$) ($j = 1,...n$), the calibration procedure estimates the β_i's ($i = 0,...5$) of the regression equation(1). The procedure uses the stepwise regression algorithm mentioned in the

previous section, with some simplifications tailored to the specific use of the model.

Since the model is to be used in system design or optimization, it must indicate values for all the system parameters in the final system. This implies that all the system parameters must appear in the expression for performance. Thus, the usual statistical significance test that is applied to decide which factors should appear in the regression equation is bypassed; instead all the factors are forced into the equation.

Further, as the outline of the optimization procedure in Chapter 2 suggests, the model needs to be accurate only in the system parameters sub-region of immediate interest. This is because of the continual process of re-calibration as the procedure explores the system parameter space. Thus at the initial stages, even if the model is not statistically significant (see [DRA66]), at some reasonable level of confidence, it will be accepted, since it is reasonable to expect that as more recalibration is done, the accuracy of the model, in local regions, will improve. Consequently, the statistical tests for significance of regression and for lack of fit using replicated observations [DRA66] are not performed. Thus the regression procedure is used solely for the least squares estimation of the β_i's.

CHAPTER 4

SYSTEM OPTIMIZATION USING THE PERFORMANCE MODEL HIERARCHY

4.1 Introduction

In the last chapter, we introduced the CPU-memory subsystem of the IBM System 360/91, as the system chosen for the case-study of the hierarchical approach to performance evaluation. We presented a functional description of the system, and two models - a control stream model and an analytic performance model - of the system. In this chapter, we discuss the construction of a hierarchy composed of these two models. We also describe a procedure that uses the hierarchy to design an optimum system.

4.2 System Optimization Objectives

The techniques described in this chapter can be used to optimize a system configuration with respect to any objective function that involves system performance. In our study, we chose cost/performance ratio as the objective function to be minimized. Other objectives that may be considered by a system designer include maximizing system performance subject to an upper bound on system cost, and minimizing system cost subject to a lower bound on system performance.

We will use the performance model hierarchy to estimate the performance component of the objective function alone. We will assume that the other components of the function can be estimated with the desired accuracy.

4.3 Application of the Hierarchical Approach

In Chapter 2, we introduced the concept of a performance model hierarchy as an efficient tool for exploring a computer system parameter space. Its efficiency arises from the two main features of the hierarchy:

1) The accuracy of performance predictions increases as we go down the hierarchy.

2) The computational cost of prediction increases as we go down the hierarchy.

The latter feature demands that a high-level model be used for performance prediction in any optimization loop, so as to keep down the computational cost of the optimization procedure. The former feature ensures the accuracy of the procedure, by providing a low-level model as the basis for re-calibration of the high-level model after every iteration of the optimization. The optimization is then repeated till the predictions of the two models converge.

4.3.1 Roles of the Models in the Hierarchy

From the discussion above, it is clear that the two models play distinct roles in the hierarchy. The control stream model serves to mark with great accuracy points on the performance surface in the six-dimensional system parameter space. The analytical model attempts to use as many of these points as necessary to obtain an approximation to the surface in a local region of the space. In fact choosing a linear, orthogonal (first-order in all its factors) analytical model, chooses a hyperplanar approximation to the performance surface in that region. It would be expected that this approximation is quite gross over large regions of the space. However, the optimization procedure relies on three factors to make this approximation palatable:

a) Constant re-calibration, i.e., using the control stream model to fill in a new point on the surface with every iteration of the optimization procedure. This causes the hyperplanar approximation of the analytical model

to change, as new points appear on the surface in the region currently being explored, or as the optimization procedure shifts to new regions in the space.

b) As the optimization procedure zeroes in on the optimum, the region of interest shrinks in size, making the hyperplanar approximation increasingly better, in that region.

c) In the optimization procedure, the hyperplane model, i.e., the analytical model,will be used primarily to indicate the preferred direction of movement on the surface toward the optimum. It will have a smaller part in deciding the magnitude of the movement in that direction.

These points will be elaborated upon in later sections. In the discussions to follow, the analytical model will also be referred to as the hyperplane model or as the hyperplanar approximation.

4.4 The Optimization Procedure

We now describe a systematic procedure for exploring a given system parameter space, to optimize an objective function that involves system performance. The description will be in terms of a general system parameter space, with the case-study being used to illustrate the concepts developed.

4.4.1 Definitions and Overview

The procedure that determines the optimum system is called the global optimization procedure. The points in the space for which the performance has been evaluated using the low-level model are called calibration points. The set of calibration points is called the calibration set. One point in this set is singled out as the reference system. This system is the focal point in the region that is currently being explored by the global procedure.

Each iteration of the global optimization procedure consists of the following steps:

a) A local optimization procedure is applied to the objective function using the current version of the analytical model. This will usually be a standard multi-dimensional, real-variable optimization procedure.

b) A movement rule is invoked to determine the new reference system from the old reference system and the prediction of the local optimization procedure.

c) A stopping rule is invoked to see if the global procedure has converged.

d) If it has not converged, a recalibration procedure is applied to the calibration set to recalibrate the analytical model at the new reference system using a subset of the calibration set called the recalibration set. This may involve evaluating the performance of the new reference system using the low-level model.

Once the global procedure has converged, a sensitivity analysis procedure is invoked to probe the region near the optimum. This is both for the purpose of identifying the true optimum in the local region to which the global procedure has converged, as well as to determine the sensitivity of the objective function to the various system parameters near the optimum.

For the first iteration of the global procedure, an initial calibration set and an initial reference system must be supplied. Figure 4.1 is a flowchart depicting the various steps of the procedure. We elaborate on each of these in the sections to follow.

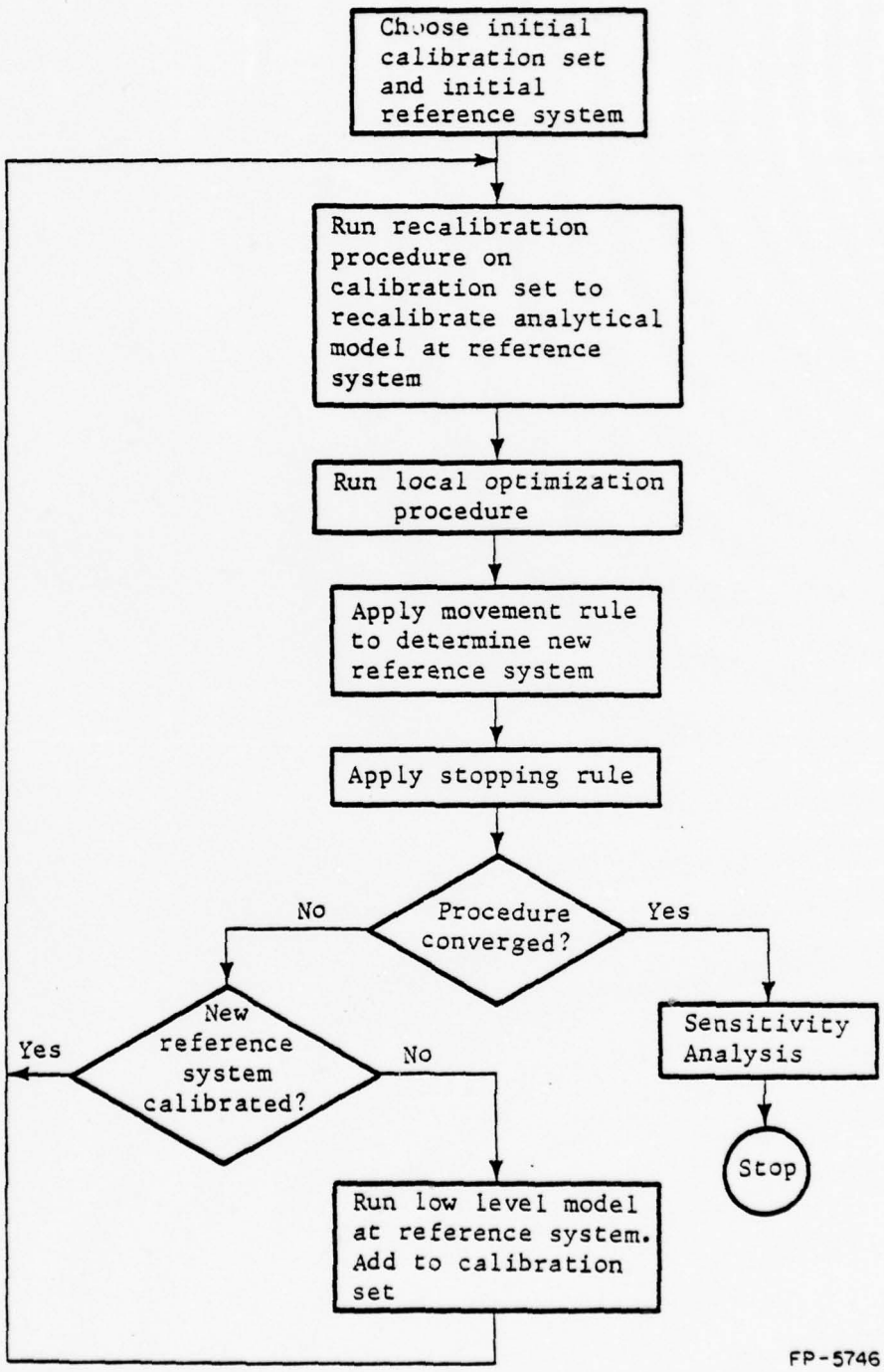


Figure 4.1 The global optimization procedure.

4.4.2 System Parameter Metrics and a Multi-dimensional Grid in the Space

Since the hyperplane model approximation to the performance surface is expected to improve for smaller regions of calibration, the concept of distance in the system parameter space becomes important. Thus, recalibration with respect to the reference system should only involve points from the calibration set that are close to that system. To quantify distance in the space, a metric has to be defined for each system parameter dimension. The metric must be chosen in such a manner that distances along individual dimensions can be combined to yield a reasonable estimate of overall distance in the space.

Often, it is most convenient to choose the metric along each system parameter dimension to reflect the maximum resolution possible in the set of values that the parameter can realistically assume. Thus, in our case-study, the metrics were chosen as shown in Table 4.1. Distance in the space is then defined as the standard n-dimensional Euclidean distance. For example, in our case-study, the distance between two systems with parameters (mc, mb, ib, fx, fℓ, ℓm) = (5, 64, 8, 3, 2, 1) and (7, 16, 32, 1, 2, 0) is:

$$[(7-5)^2 + (4-6)^2 + (5-3)^2 + (1-3)^2 + (2-2)^2 + (0-1)^2] = \sqrt{17} \text{ metric units.}$$

By this definition of a metric along each dimension, we have laid a multi-dimensional grid on the system parameter space, with points of the grid spaced one metric unit along each dimension. Each grid point now represents a realistic system configuration. The goal of the optimization procedure is to identify the grid point which represents the optimum system.

4.4.3 The Initial Calibration Set

If the global optimization procedure can be shown to converge to the global optimum system regardless of the starting point, the initial

Table 4.1 - Metrics for the System Parameter Dimensions

System Parameter	Value in natural units	Value in metric units
mc	mc CPU cycles	mc
mb	mb banks	log mb
ib	ib buffers	log ib
fx	fx architectural units (see section 3.4.10)	fx
fl	fl architectural units	fl
lm	lm architectural units	lm

reference system could be arbitrarily chosen. Since this may not be the case, the choice of the initial reference system may have to be based on intuition or experience, or be fixed by other cost or performance constraints. In our case-study we arbitrarily chose the point that matches the existing 360/91 as the initial reference system. This system shall henceforth be referred to as the normal system.

An initial calibration set must then be chosen around the reference system. Since this is the calibration set for the first iteration of the global optimization procedure, it must be chosen so as to yield enough information for a reasonable fit of the hyperplane model in the region around the initial reference system. This implies that there should be points on either side of the reference system along each dimension. In our case-study, we chose a very sparse set consisting of systems each of which had a change in only one dimension from the reference system. Table 4.2 lists the initial calibration set chosen for the case-study. This is repeated for $\ell_m = 0$ and $\ell_m = 1$ as will be discussed in Sec. 4.5.1. The parameter values are given in the natural units listed in Table 4.1. This will be the practice through the rest of this report.

The control stream model is then used to predict system performance at all the points of the initial calibration set. These points on the performance surface are then used to calibrate the analytical performance model to be used in the first iteration of the global optimization procedure. The calibration is done as described in Sec. 3.4.2.1.

4.4.4 The Movement Rule

Since the local optimization procedure is supplied with a very approximate evaluation of the objective function, viz, one involving the hyperplane model, its predictions must be used carefully to prevent the global

Table 4.2 - Initial Calibration Set and Reference System

System	System Parameters				
	mc	mb	ib	fx	fl
Reference System 1	12	16	8	1	3
2	6	16	8	1	3
3	18	16	8	1	3
4	12	8	8	1	3
5	12	32	8	1	3
6	12	16	8	2	3
7	12	16	8	3	3
8	12	16	8	1	1
9	12	16	8	1	2
10	12	16	4	1	3
11	12	16	16	1	3

procedure from making wild excursions. For a simple example to illustrate this, consider the unidimensional performance function P(S) in Figure 4.2. The analytical model for this dimensionality is a straight line. When the model is M_1 about the reference system S_1, its range of validity R_1, is much larger than the range R_2, of the model M_2 about the reference system S_2. Thus movement must be much more restricted when the optimization uses the model M_2 than when it used M_1. Even when using the model M_1, movement must be somewhat restricted to prevent it from going outside the range R_1.

In the optimization of the case-study system the following movement rule was adopted. When the local optimization procedure arrives at an optimum based on the current analytical model, the reference point is shifted one grid point (in metric units) along each dimension in the direction that the predicted optimum is located with respect to this reference point. This is done regardless of the magnitude of the distance between the reference point and the predicted optimum.

The restriction of the movement to one grid unit prevents excursions beyond the range of validity of the analytical model. However the enforced movement, regardless of the magnitude predictions of the local optimization procedure, forces the procedure to make a rough exploration of as much of the objective function surface as possible in the initial stages, before zeroing in on a particular region as the most promising one for finer exploration. Examples of the application of the movement rule are given in Table 4.3. It should be noted that the movement along the mc dimension is three metric units in the first example and one metric unit in the second. This is a consequence of the adaptively varying metric unit adopted for the mc dimension, to be discussed in Sec. 4.5.2.

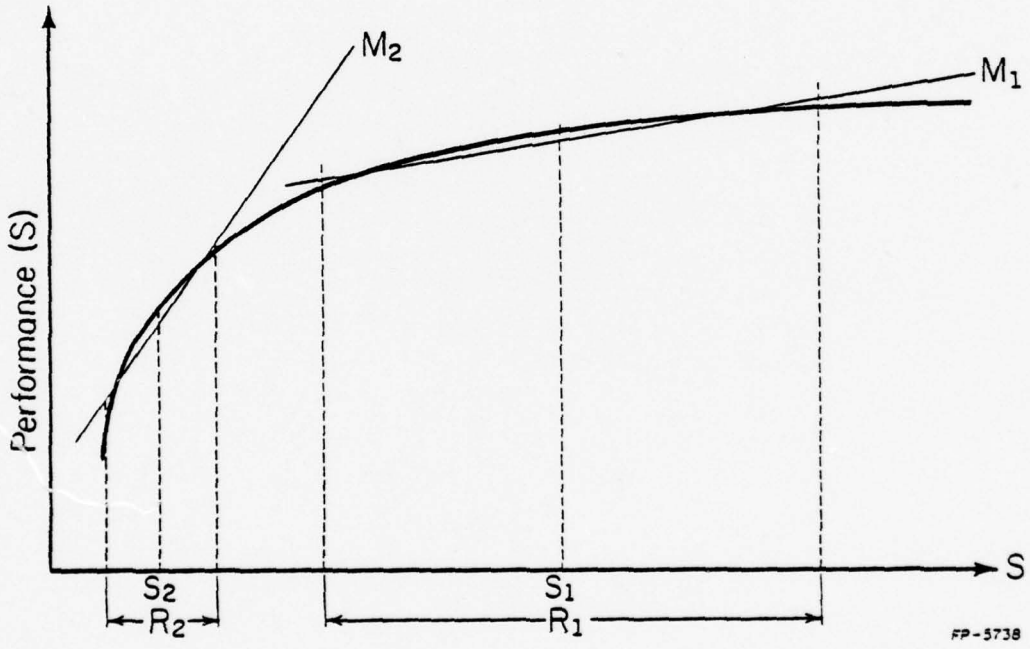


Figure 4.2 Validity of the analytical model in different regions.

Table 4.3 - Application of the Movement Rule

	System Parameters				
	mc	mb	ib	fx	fl
Starting reference system	12	16	8	1	3
Optimum system predicted by hyperplane model	9.6	63.9	31.9	2.9	0.7
New reference system	9	32	16	2	2

Example 1.

Starting reference system	6	32	16	2	1
Optimum system predicted by hyperplane model	4.2	38.4	12.7	3.5	1.6
New reference system	5	64	8	3	2

Example 2.

4.4.5 The Stopping Rule

When successive iterations of the global procedure cause an oscillation of the reference point between two calibration systems, the procedure is deemed to have converged and is stopped. To understand this, let us consider the cost/performance function of one system parameter in Figure 4.3. Suppose the n-th iteration of the procedure using model M_1, at the reference system S_1 causes a movement of the reference point to S_2. Let the recalibrated model at S_2 be M_2 which, on the $(n+1)$st iteration dictates a movement of the reference point back to S_1. If S_2 already existed in the calibration set at the n-th iteration, the situation now is exactly the same as at the n-th iteration. Thus the $(n+2)^{nd}$ iteration must cause the reference point to move back to S_2. The procedure would thus oscillate forever between S_1 and S_2. It is therefore stopped.

4.4.6 Heuristic Algorithms for Recalibration of the Analytical Model

For each iteration of the global optimization procedure, the reference point is moved to a new system by the application of the movement rule described in Sec. 4.4.4. If the new reference system is not a calibration point, the low level model is used to accurately evaluate the performance at this system, which is then added to the calibration set. The analytical model is then recalibrated at this reference system choosing an appropriate recalibration set. This possibly new hyperplanar approximation is used in the next iteration of the global optimization procedure. In this section, we discuss algorithms for performing this recalibration.

4.4.6.1 Calibration requirements

For the hyperplane model to be a good approximation, information local to the region around the reference system must be used in the

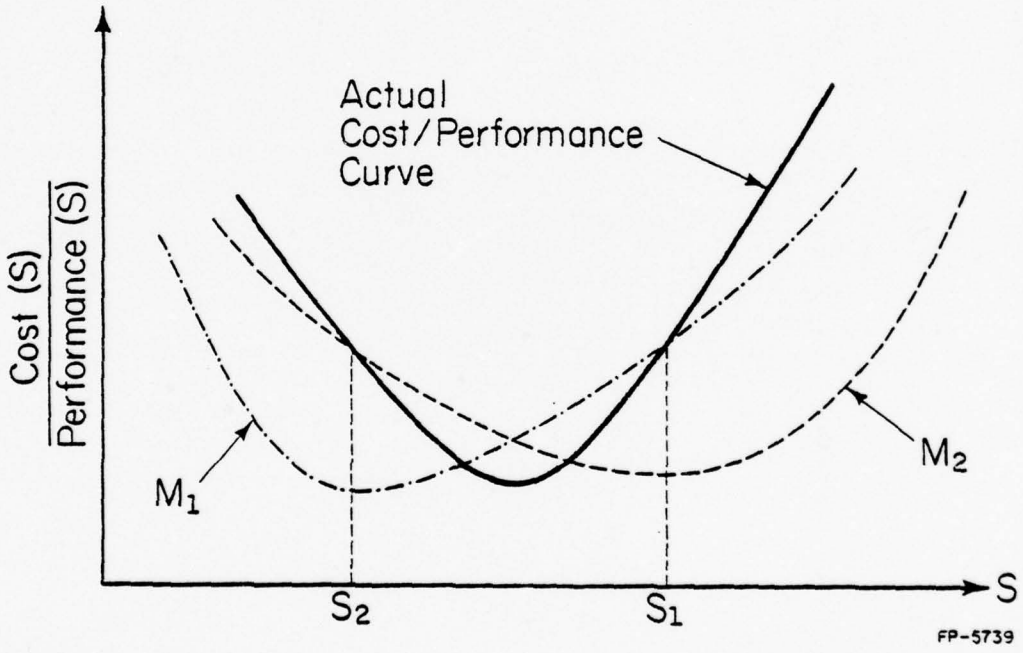


Figure 4.3 Oscillation between two reference systems.

recalibration. Thus the set of points chosen for recalibration must satisfy the following two criteria.

1) Along those dimensions for which the reference system does not have an extreme value, i.e., calibration points exist on either side of the reference system along that dimension, the point closest to the reference system each direction along that dimension must be in the set.

2) Along dimensions for which the reference system does have an extreme value, i.e., no calibration point has a value beyond that of the reference system in one direction along that dimension, the point closest to it in the other direction along that dimension must be in the set.

For example, the minimum recalibration set for the grid points of Table 4.2, with system 0 as reference, will not include systems 7 and 8, since system 0 has extreme values along the fx and fℓ dimensions and systems 7 and 8 are not the closest ones to 0 along those dimensions.

These two criteria are used by the two recalibration algorithms to be discussed now.

4.4.6.2 Multi-dimension Recalibration Algorithm

In this algorithm, all the calibration points are ordered in increasing order of their distance from the reference system. Systems for the recalibration set are chosen in this order, until the above two criteria are satisfied for all the dimensions. At this stage, any other points that are at the same distance from the reference system as the point in the recalibration set at the maximum distance from the reference system are also included in the set. This recalibration set is then used by the multi-dimensional regression procedure, described in Sec. 3.4.2.1, to estimate the β_i's of the analytical model. If the cardinality of the set is insufficient for the regression procedure to estimate all the model parameters,

further points are added to the set in the increasing order of their distance from the reference system, until the cardinality is sufficient.

Table 4.4 shows the results of an application of this recalibration algorithm to the case-study system. Notice that the grid unit distance in the mc dimension is six cycles in this example. The reason for this is explained in Sec. 4.5.2. The lower half of Table 4.4 shows percentage errors between the observed throughput values from the control stream model and the fitted throughput values from the analytical model. Thus, for the systems in the recalibration set in this example, this algorithm achieves an upper bound of 2% on the absolute regression error.

It should be observed that the algorithm as described chooses the smallest local region containing the calibration points needed to satisfy the criteria of Sec. 4.4.6.1, This is the implementation of the philosophy of using the analytical model to make only local predictions. Including more systems than the above minimum, would decrease the locality of the model and increase the regression error bound of the algorithm.

4.4.6.3 Individual Dimension Recalibration Algorithm

In this algorithm, system clusters are identified separately for each dimension. These clusters are then used to calculate the slope, i.e., the β_i, along that dimension, independently of the other dimensions.

The clusters are formed in the following manner. For each dimension, the point or points closest to the reference system along that dimension, according to the two criteria of Sec. 4.4.6.1 are first included in the cluster. If there is more than one candidate for these nearest neighbor points, the one that is the closest to the reference system in the n-dimension Euclidean sense defined in Sec. 4.4.2 is selected. Next, all

Table 4.4 - Multi-dimension Recalibration Algorithm

```
Calibration set: 00 02 03 04 05 06 07 08 09 10 11 27
Loopmode: 1 (on)
Reference system: 27
mc grid metric:  6 CPU cycles

Distance of systems from reference system:
System       mc       mb       ib       fx       fl       lm     Norm^2
    27     0.00     0.00     0.00     0.00     0.00     0.00       0.00
     5     0.33     0.00    -1.00    -1.00     1.00     0.00       3.11
     6     0.33    -1.00    -1.00     0.00     1.00     0.00       3.11
     9     0.33    -1.00    -1.00    -1.00     0.00     0.00       3.11
    11     0.33    -1.00     0.00    -1.00     1.00     0.00       3.11
     0     0.33    -1.00    -1.00    -1.00     1.00     0.00       4.11
     7     0.33    -1.00    -1.00     1.00     1.00     0.00       4.11
     8     0.33    -1.00    -1.00    -1.00    -1.00     0.00       4.11
     2    -0.67    -1.00    -1.00    -1.00     1.00     0.00       4.44
     3     1.33    -1.00    -1.00    -1.00     1.00     0.00       5.78
     4     0.33    -2.00    -1.00    -1.00     1.00     0.00       7.11
    10     0.33    -1.00    -2.00    -1.00     1.00     0.00       7.11

Re-calibration set: 27  5  6  9 11  0  7  8  2

Hyperplane model coefficients:
  Constant        mc        mb        ib        fx        fl
   0.28991  -0.01287   0.01939   0.00425   0.01094   0.00761

Error in hyperplane model predictions of throughput:
System  Observed   Predicted    %Error
    27   0.30965     0.31226     -0.84
     5   0.28156     0.27895      0.93
     6   0.26681     0.27050     -1.38
     9   0.24924     0.25195     -1.09
    11   0.26642     0.26381      0.98
     0   0.25669     0.25956     -1.12
     7   0.28458     0.28144      1.11
     8   0.24699     0.24434      1.08
     2   0.33764     0.33677      0.26
     3   0.20186     0.18235      9.66
     4   0.22391     0.24018     -7.27
    10   0.19933     0.25531    -28.09
```

points within the hypercube defined by the reference system and the nearest neighbor points are included in the cluster. Thus the cluster may contain points that differ from the reference system in dimensions other than the one under consideration. Let there be m such dimensions. The points in the cluster are used by the multi-dimensional regression procedure of Sec. 3.4.2.1 to fit a (m+1) factor regression equation with the dimension of interest being forced to appear in the regression equation. That coefficient alone is taken from the resultant regression equation, and used as an estimate for the β_i along that dimension.

This procedure is repeated for each of the dimensions. Once the β_i's have been estimated for each dimension, β_0 is calculated by forcing the analytical model to exactly match the performance of the reference system.

Table 4.5 shows the results of the application of this algorithm to the same set of calibration points as in Table 4.4. For example, along the mc dimension, systems 5, 6, 9, and 11 are the nearest neighbors of the reference system 27 on the right, while system 2 is the nearest neighbor on the left. These define a hypercube with dimension ranges (-0.67 to 0.33, -1 to 0, -1 to 8, -1 to 0, 0 to 1) in metric units. Since system 0 is inside this hypercube, it is included in the cluster for recalibration of the mc dimension.

It can be seen that the precision of this recalibration algorithm is far worse than that of the multi-dimension algorithm. In this example, the individual dimension algorithm achieved an upper bound of 10% on the absolute error for systems that occur in some cluster, as compared to 2% by the multi-dimension algorithm. In fact in all the examples that were

Table 4.5 - Individual Dimension Recalibration Algorithm

```
Calibration set: 00 02 03 04 05 06 07 08 09 10 11 27
Loopmode: 1 (on)
Reference system: 27
mc grid metric: 6 CPU cycles

Distance of systems from reference system:
System       mc       mb       ib       fx       fl       lm     Norm^2
    27     0.00     0.00     0.00     0.00     0.00     0.00       0.00
     5     0.33     0.00    -1.00    -1.00     1.00     0.00       3.11
     6     0.33    -1.00    -1.00     0.00     1.00     0.00       3.11
     9     0.33    -1.00    -1.00    -1.00     0.00     0.00       3.11
    11     0.33    -1.00     0.00    -1.00     1.00     0.00       3.11
     0     0.33    -1.00    -1.00    -1.00     1.00     0.00       4.11
     7     0.33    -1.00    -1.00     1.00     1.00     0.00       4.11
     8     0.33    -1.00    -1.00    -1.00    -1.00     0.00       4.11
     2    -0.67    -1.00    -1.00    -1.00     1.00     0.00       4.44
     3     1.33    -1.00    -1.00    -1.00     1.00     0.00       5.78
     4     0.33    -2.00    -1.00    -1.00     1.00     0.00       7.11
    10     0.33    -1.00    -2.00    -1.00     1.00     0.00       7.11

Individual dimension clusters:
Parm    Systems
  mc    0   2   5   6   9  11  27
  mb    0   5   6   9  11  27
  ib    0   5   6   9  11  27
  fx    0   5   6   7   9  11  27
  fl    0   5   6   8   9  11  27

Hyperplane model coefficients:
  Constant        mc        mb        ib        fx        fl
   0.20595  -0.01307   0.02859   0.01345   0.01395   0.00485

Error in hyperplane model predictions of throughput:
System  Observed   Predicted    %Error
     0   0.25669     0.23232      9.49
     2   0.33764     0.31074      7.97
     3   0.20186     0.15390     23.76
     4   0.22391     0.20373      9.01
     5   0.28156     0.26091      7.33
     6   0.26681     0.24627      7.70
     7   0.28458     0.26022      8.56
     8   0.24699     0.22262      9.87
     9   0.24924     0.22747      8.73
    10   0.19933     0.21887     -9.80
    11   0.26642     0.24577      7.75
    27   0.30965     0.30960      0.00
```

tried, this method had a larger error bound. The reason would appear to be that the individual-dimension algorithm neglects some factor interactions because of the piecemeal approach, whereas the multi-dimension approach tries to fit an overall model that approximates the interactions as best as it can. Consequently, the multi-dimension algorithm was the one chosen for the recalibration procedure of the analytical model.

4.4.7 Bounding the Error of the Analytical Model

To control the global optimization procedure even further, an upper bound can be computed for the prediction error of the analytical model at each iteration. If the error exceeds the bound, the predictions of the procedure can be further checked or the procedure itself modified. We now develop an expression for a conservative bound, that can be used for this error checking.

Let us represent points in the system space by the vectors $S = (s_1, \ldots s_5)$. Let the actual performance and cost surfaces be $P(S)$ and $C(S)$, respectively, both of which are positive. Let the objective function, which is to be minimized be $f(S) = C(S)/P(S)$. Let the reference system, at which the local optimization procedure begins, be $S_r = (s_{r1}, \ldots s_{r5})$. Then the direction of movement of the procedure should be given by the gradient of f at S_r. Thus, along the i-th dimension,

$$\nabla f_i(S_r) = \frac{\partial f(S)}{\partial s_i}\Big|_{S_r}$$

$$= \frac{\partial}{\partial s_i}\left(\frac{C(S)}{P(S)}\right)\Big|_{S_r}$$

$$= \frac{C(S_r) \cdot \frac{\partial P(S)}{\partial s_i}\Big|_{S_r} - P(S_r) \cdot \frac{\partial C(S)}{\partial s_i}\Big|_{S_r}}{[P(S_r)]^2}$$

$$= \frac{C(S_r) \cdot \nabla P_i(S_r) - P(S_r) \cdot \nabla C_i(S_r)}{[P(S_r)]^2}$$

where $$\nabla P_i(S_r) = \frac{\partial P(S)}{\partial s_i}\Big|_{S_r}$$

and $$\nabla C_i(S_r) = \frac{\partial C(S)}{\partial s_i}\Big|_{S_r} .$$

Assuming that the cost function is orthogonal, but not necessarily first order, in its parameters, let

$$\nabla C_i(S_r) = k_i(s_{ri}) .$$

In optimization using the analytical model, the cost function is assumed to be exact. However $\nabla P_i(S_r)$ is approximated by β_i, the i-th coefficient of the regression expression. Assuming that $P(S_r)$ is predicted exactly by the analytical model, the approximation to $\nabla f_i(S_r)$ is

$$\hat{\nabla} f_i(S_r) = \frac{C(S_r) \cdot \beta_i - P(S_r) \cdot k_i(s_{ri})}{[P(S_r)]^2} .$$

Thus, the bound on the error of β_i is such that $\nabla f_i(S_r)$ and $\hat{\nabla} f_i(S_r)$ have the same sign, so as to cause the local optimization procedure to move in the correct direction along the i-th dimension. Applying theorem A.1 (see Appendix A), this is satisfied if

$$\left|C(S_r) \cdot \nabla P_i(S_r) - C(S_r) \cdot \beta_i\right|$$

$$< \left|C(S_r) \cdot \nabla P_i(S_r) - P(S_r) \cdot k_i(s_{ri})\right|,$$

which reduces to

$$\left|\nabla P_i(S_r) - \beta_i\right| < \left|\nabla P_i(S_r) - \frac{P(S_r)}{C(S_r)} \cdot k_i(s_{r2})\right| . \qquad (4.1)$$

However, to use this bound on the error in β_i, the value of $\nabla P_i(S_r)$ is required. This can be approximated by considering the broken hyperplane approximation to the surface. For example, consider the one-dimensional analytical model in Figure 4.3. Let m_1 and m_2 be the slopes of the two segments of the broken straight line approximation to the curve, that pass through S_r. Then $\nabla P_i(S_r)$ may be roughly bounded by:

$$m_1 \geq \nabla P_i(S_r) \geq m_2.$$

If, for $\nabla P_i(S_r)$ in inequality 4.1, we substitute an estimate based on m_1 and m_2, the bound can be used in practice. The estimate can be chosen so as to develop either a worst-case bounding condition or an average bounding condition. Thus the worst case inequality that bounds β_i is:

$$\max(|m_1 - \beta_i|, |m_2 - \beta_i|)$$

$$< \min(|m_1 - \frac{P(S_r) \cdot k_i(s_{ri})}{C(S_r)}|,$$

$$|m_2 - \frac{P(S_r) \cdot k_i(s_{ri})}{C(S_r)}|) .$$

A more optimistic bounding inequality would be:

$$|\text{mean}(m_1, m_2) - \beta_i|$$

$$< |\text{mean}(m_1, m_2) - \frac{P(S_r) \cdot k_i(s_{ri})}{C(S_r)}|$$

If we now allow for an error in the model prediction of performance $mP(S_r)$ the bounding condition is:

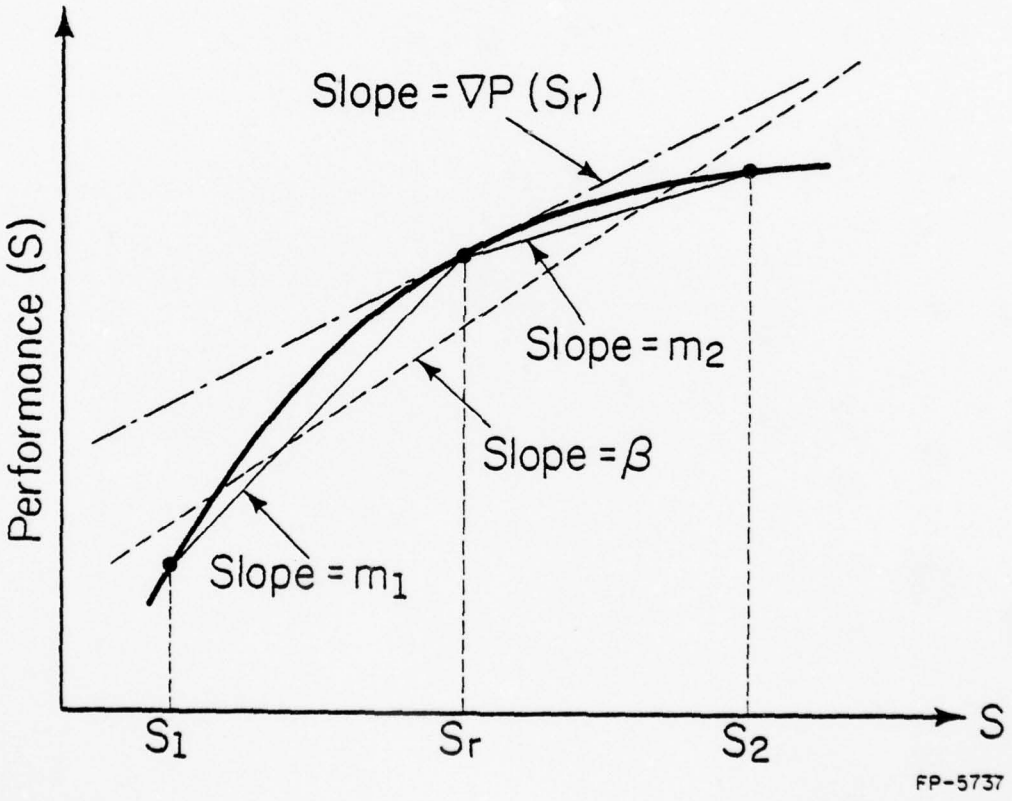


Figure 4.4 Bounding the slope of the performance curve.

$$|\text{mean}(m_1, m_2) - \beta_i|$$

$$< \left| \text{mean}(m_1, m_2) - \frac{k_i(s_{ri})}{C(S_r)} \cdot \text{mean}(P(S_r), mP(S_r)) \right|$$

For every recalibration of the analytical model, the error bound conditions can be calculated as above, for each dimension. The confidence in prediction is high for those parameters that satisfy the inequality. For those that do not, one of the following corrective steps may be taken:

1) The direction opposite to that of the prediction can be examined.

2) The recalibration set can be changed to eliminate some possibly misleading calibration points.

4.4.8 Sensitivity Analysis

When the global procedure has converged, sensitivity analysis must be conducted in the region around the predicted optimum for the following reasons:

a) To precisely identify the optimum in the region of oscillation.

b) To determine the sensitivity of performance, and hence the objective function, to small changes from the optimum system. This may indicate changes that can be made in the system design, for a very small sacrifice in performance or cost. Such changes may be attractive to meet other design objectives. Since accurate predictions of performance are needed to meet the above objectives, the low-level model must be used for performance evaluation at this stage.

4.4.8.1 Exhaustive Sensitivity Analysis

This procedure evaluates the performance, and hence the objective function, at all the grid point neighbors of the predicted optimum. This is repeated until a system which is better than all its neighbors is found.

This system is a local optimum. For example, if the global procedure settles on (5, 64, 8, 2, 2) as the optimum system, this procedure would evaluate systems with all combinations of the following parameter values:

mc: 4, 5, 6

mb: 32, 64, 128

ib: 4, 8, 16

fx: 1, 2, 3

fℓ: 1, 2, 3

This would require the evaluation of $3^5 = 243$ systems, for the first analysis of sensitivity.

4.4.8.2 Single Parameter Sensitivity Analysis

To avoid the large number of expensive calls on the low-level model required by the exhaustive procedure, a single parameter approach to sensitivity was adopted. In this procedure, the neighboring systems along a single dimension are compared with the predicted optimum. If one of these is better, it is made the new optimum and its new neighbor along the same dimension is examined. This procedure is repeated until no improvement can be made along that dimension. Each of the other dimensions is then examined individually, holding all the other parameters constant. If after one pass through all the dimensions, no change was made to the optimum, the procedure is stopped. If there were any changes, another pass is made through all the dimensions.

Examples of this procedure are given in the next chapter.

4.4.9 Efficiency of the Optimization Procedure

Since most of the cost of using the model hierarchy, is in the use of a low-level model for performance prediction, the procedure can be compared

with other procedures by comparing the number of calls on the low-level model. Two benchmark procedures that bound the efficiency are now defined.

a) The Ideal Procedure: In this procedure, the optimum system is already known. The cost of the procedure is then associated with the sensitivity analysis around the optimum, which must still be conducted for the second reason in Sec. 4.4.8. Thus we will assume that the minimum cost that the system designer must bear is the cost of the sensitivity analysis in the ideal procedure. The efficiency of the ideal procedure is defined as 1. All procedures will be compared with the ideal procedure to estimate their efficiency.

b) The Grid Evaluation Procedure: In this procedure, the low-level model is used to evaluate the performance of all the grid points in the given region. The best system in that region is then chosen. For our case-study, we will assume that the region of interest is bounded by the minimum and maximum along each dimension, that was ever used as a calibration point by the optimization procedure. For example, if the maximum excursions along each dimension were:

mc: 4 to 12 (9 grid points)

mb: 16 to 128 (4 grid points)

ib: 4 to 32 (4 grid points)

fx: 1 to 3 (3 grid points)

fℓ: 1 to 3 (3 grid points),

the grid evaluation procedure would make 1296 calls on the low-level model to evaluate all the grid points in this region.

With these definitions, the efficiency of the optimization procedure is defined as

$$\eta = \frac{\text{Number of calls to the low-level model by the ideal procedure}}{\text{Number of calls to the low-level model made by the optimization procedure.}}$$

A lower bound on achievable efficiency is defined as

$$\eta_L = \frac{\text{Number of low-level model calls by the ideal procedure}}{\text{Number of low-level model calls by the grid evaluation procedure}} .$$

Both these estimates are used in the next chapter.

4.5 Adaptation of the General Procedure to the Case-Study

In the previous section, we described a general procedure for finding the optimum point in a given system parameter space. In this section, we discuss some of the choices and assumptions made in applying this procedure to our case-study.

4.5.1 Continuous vs. Non-continuous System Parameters

The choice of a grid on the space was dictated by the desirability of examining only realistic computer systems. The grid points represent only such systems. However, for some parameters, points other than grid points also represent possible systems. Thus mb or ib can conceivably have a value that is not a power of two. A non-integral value for mc could be achieved by a finer division of the cycle time. A value of 2.5 for the fixed point unit architecture may represent a design that is a compromise between the pipelined architecture and dataflow architecture described in Sec. 3.4.6.3. Thus these system parameters can be approximated by real values. However, in the local optimization procedure, the ℓm parameter is strictly a Boolean parameter. That is, a system either does or does not have loop-mode. Thus non-boolean values for ℓm would be unrealistic.

In view of the above, the global optimization was split into two parts - finding an optimum system on the $\ell m = 0$ hyperplane and another on the $\ell m = 1$ hyperplane. On each of these hyperplanes, a standard multi-dimensional, real variable optimization procedure was used as the local optimization procedure to determine the other five parameters of interest. The better of the optima on the two hyperplanes is then identified as the optimum system.

In an early version of the optimization procedure, the initial calibration set for optimization on the $\ell m = 0$ hyperplane consisted of only one system - the normal system with loop-mode turned off. Projections of the initial calibration set on the $\ell m = 1$ hyperplane (see Table 4.2) onto the $\ell m = 0$ hyperplane were used by the procedure in its initial calibration. This approach is a logical extension of the orthogonality assumption to the ℓm dimension. However, the errors in this assumption were so large as to cause the procedure to move very erratically on the $\ell m = 0$ hyperplane. Consequently, this approach was abandoned and an entire initial calibration set as in Table 4.2, was used on the $\ell m = 0$ hyperplane as well.

4.5.2 Adaptive Metric for the mc Dimension

Since the range of the mc parameter is considerably larger than the ranges of the other parameters, a variable metric was chosen for that dimension. If the metric were chosen as one CPU cycle, it was expected that the global optimization procedure would cause very small incremental moves in the mc dimension. To avoid this, the metric was chosen as six cycles initially. This explains the settings of 6 and 18 for mc in the initial calibration set of Table 4.2. However, if this large metric value were retained at later stages in the optimization, systems with large

differences in mc from the reference system would still be included in the recalibration set, because the large mc metric makes them appear closer (in metric units) to the reference system. Thus the locality of the analytical model calibration is lost. This effect was actually observed in an early version of the optimization procedure.

To compromise between these two opposing needs, the following mc movement and metric reduction rules were adopted:

1) If the movement dictated by the optimization along the mc dimension is larger than the current mc metric, the reference point is moved just 1 metric unit along the mc dimension in that direction. If the movement dictated is less than one metric, the reference point is moved to the first integral value of mc, beyond the predicted optimum from the reference point.

2) The initial value of the mc metric is six cycles. This value may be decreased once every three iterations of the optimization procedure and is held constant in between. The new value for the metric is chosen as follows:

If the maximum movement in the last three iterations was equal to the old metric value (*it cannot be greater*), the metric value is decreased by 1. If the maximum movement in the last three iterations was less than the old metric value, the metric value is reduced to the value of that maximum. The metric is, however, never reduced below 1.

3) If oscillation occurs between two calibration points that are more than one cycle apart in the mc dimension, the oscillation is broken by reducing the metric by 1.

By the above procedure, the mc metric will eventually be reduced to 1, the resolution required of the optimization procedure. The metric reduction thus progressively expands the mc dimension so as to focus attention on the region of interest, by excluding points far away from having any effect on the calibration. The rate of this expansion is tied to the rate at which the optimization procedure seems to converge - which is indicated by the magnitude of the moves that it dictates. Some examples of the application of the movement rule for the mc dimension are given in Table 4.6.

4.5.3 Feasibility Checking

Since only a restricted region of the system space can be simulated by the control stream model, the local optimization procedure must be restricted to this feasible region of the space. For example, fixed point unit architectures of type 1, 2 and 3 are the only configurations allowed in the simulator. However, we found that enforcing very strict feasibility checks such as $1 \leq fx \leq 3$, allowed the local optimization very little freedom of movement and caused it to stagnate at some feasibility region boundaries. To counter this we initially designated the feasible region to be defined by the inequalties:

$$0.5 \leq mc \leq \infty$$

$$0.5 \leq mb \leq \infty$$

$$0.5 \leq ib \leq \infty$$

$$0.5 \leq fx \leq 3.5$$

$$0.5 \leq f\ell \leq 3.5$$

Table 4.6 - Movement in the mc Dimension

Example	1	2	3	4
Current mc grid metric	6	3	2	1
mc component of starting reference system	24	9	6	6
mc component of optimum system predicted using hyperplane model	10.2	7.7	3.6	6.7
mc component of new reference system	18	7	4	7

However, as experience and insight were gained in the use of the optimization procedure, it was decided that applying only the above checks for feasibility allowed the local optimization procedure too much freedom of movement, resulting in it exploring regions far beyond the region of validity of the analytical model. For example, on one iteration, starting at a reference system with ib = 8, the procedure reached an optimum system with ib = 160. Since the linear dependence of performance on ib resulted in an exaggerated value of performance at ib = 160, changes in the other parameters did not appear very cost-effective at that high level of performance, and were therefore ruled out by the procedure.

To avoid the pitfalls described above and to restrict the movement of the procedure to the local region of validity of the analytical model an additional feasibility region was delineated. Initially this region was defined to be bounded by grid points one metric unit away from the reference system along each direction of each dimension. Thus if the reference system were (5, 32, 8, 2, 2), with the mc grid metric at two cycles, the feasible region was defined by:

$$3 \leq mc \leq 7,$$

$$16 \leq mb \leq 64,$$

$$4 \leq ib \leq 16,$$

$$1 \leq fx \leq 3$$

and $$1 \leq f\ell \leq 3.$$

However, this was found to cramp the movement of the procedure severely, and the feasibility region boundary was extended to include grid points two metric units from the reference system along each direction of each dimension.

For the example above, this defines the feasible region by:

$$1 \leq mc \leq 9,$$

$$8 \leq mb \leq 128,$$

$$2 \leq ib \leq 32$$

$$0.5 \leq fx \leq 3.5 \text{ and}$$

$$0.5 \leq f\ell \leq 3.5.$$

Notice that the bounds imposed by the simulation range of the control stream model are combined with the bounds restricting movement, to arrive at the overall feasible region bounds.

CHAPTER 5

DESCRIPTION OF EXPERIMENTS AND ANALYSIS OF RESULTS

5.1 Introduction

In Chapter 4, we described a procedure that uses the performance model hierarchy to optimize the design of a system, with respect to an objective function that involves system performance. We also outlined some of the special assumptions made in applying this procedure to the case-study system described in Chapter 3. In this chapter, we describe the experiments conducted in this case-study, and analyse the results.

5.2 Software Used

In this section, we list the software packages that were used in the case-study. Except where otherwise noted, all the software was run on the DEC-10 system at the Coordinated Science Laboratory of the University of Illinois.

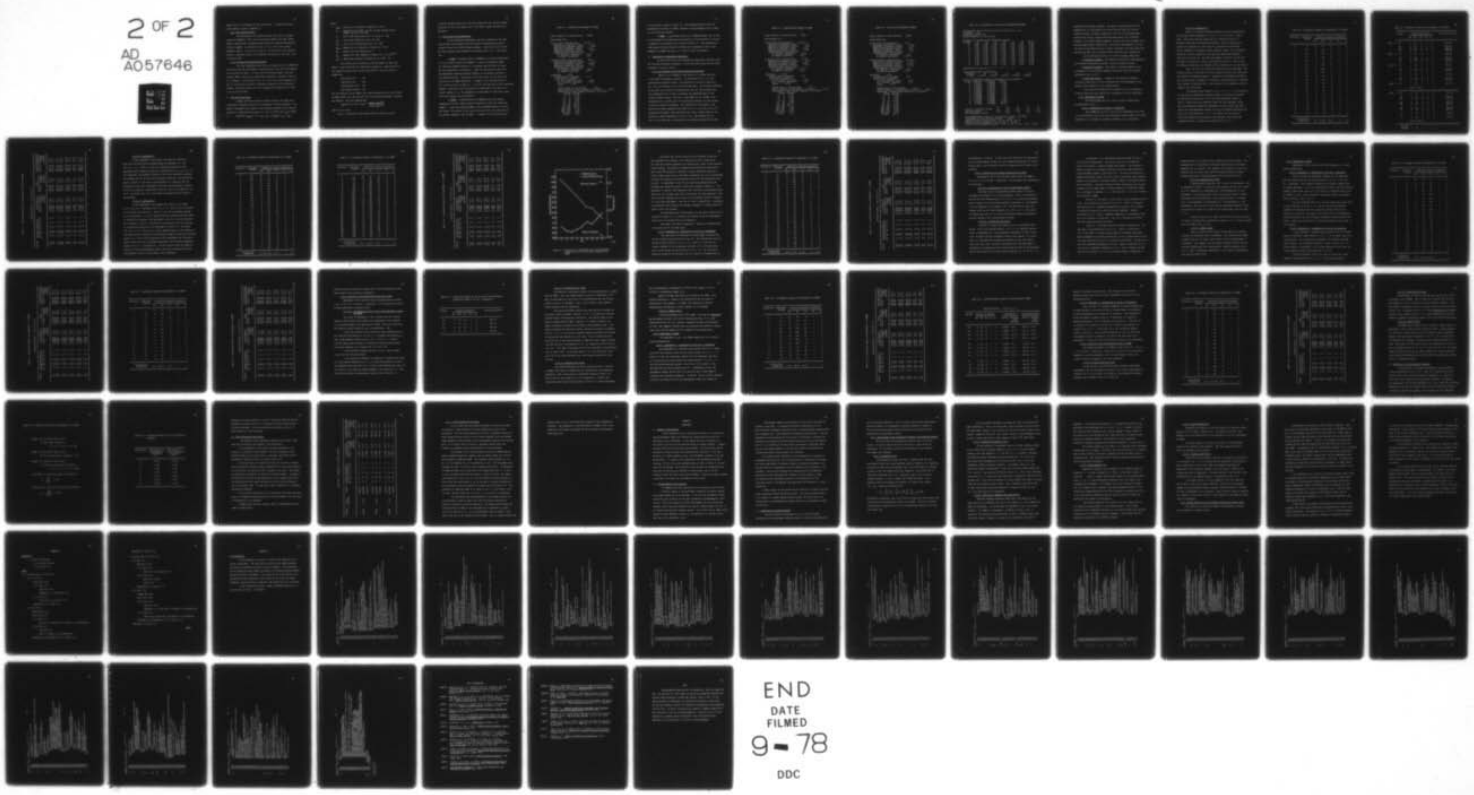
5.2.1 The Control Stream Model

To gather the program traces that generated the control streams, a modified version of the TRACE-360 program, purchased from the University of Waterloo, was used. This program, which was run on the IBM 360/75 system at the Computing Services Office of the University of Illinois, outputs the dynamic instruction execution trace as well as the memory locations referenced by the instructions of the program being traced. The conversion of the traces to control streams was done by a program written in SAIL on the DEC-10.

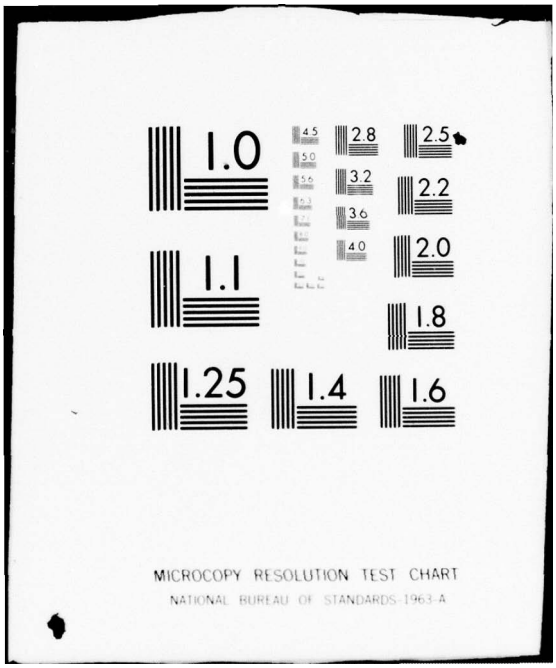
The simulator of the model was coded in SIMULA-10. Its length is about 900 lines of code. Execution times of the simulator on the KI-10 CPU

AD-A057 646 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 5/1
COMPUTER SYSTEM DESIGN USING A HIERARCHICAL APPROACH TO PERFORM--ETC(U)
OCT 77 B KUMAR DAAB07-72-C-0259
UNCLASSIFIED R-799 NL

2 OF 2
AD
A057646



END
DATE
FILMED
9-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ranged from 12 to 40 minutes for the traces used. A listing of the simulator program is given in Appendix B.

5.2.2 The Analytical Model

The calibration of the analytical model was done by a program written in SIMULA-10. This calls the procedure RLSEP of the IMSL library [IMS75], which selects a regression model using the stepwise algorithm described in [DRA66]. As described in Sec. 3.5.2.1, all the five system parameter factors - mc, mb, ib, fx, and fℓ - are forced into the regression equation. Execution times of the calibration program were less than 1 sec. on the KI-10 CPU.

5.2.3 The Local Optimization Procedure

The local optimization was done by a program written in FORTRAN-10. This first reads the coefficients of the analytical model and the parameters of the system cost model. It then calls the procedure ZXMIN of the IMSL library [IMS75], which uses a quasi-Newton algorithm to minimize a function of n variables. As described in Sec. 4.5.1, the five system parameters mc, mb, ib, fx and fℓ are treated as continuous variables by the optimization program. Execution times of the local optimization program were less than 1 sec.on the KI-10 CPU.

5.3 The System Cost Model

A simple system cost model was chosen, which at the same time, incorporates reasonably realistic cost functions of system parameters. The model is orthogonal with respect to the individual parameters and expresses system cost, normalized with respect to a cost of 100 for the actual 360/91 as

$$C(\text{system}) = F_{CPU}(C_0 + C_i + C_{fx} + C_{f\ell}) + F_{mem}(M_0 + C_{mc} + C_{mb}).$$

where:

F_{CPU} : CPU cost as a fraction of system cost = 0.57.

C_0 : fixed cost of the CPU = 15 (for the Main Storage Control Element section of the CPU)

C_i : cost of the instruction unit = $9 + ib/8 + 5 \cdot \ell m$.

C_{fx} : cost of the fixed point unit = $24 + fx^2$

$C_{f\ell}$: cost of the floating point unit = $36 + f\ell^2$.

F_{mem} : Memory cost as a fraction of system cost = 0.43.

M_0 : fixed cost of the memory unit = 2

C_{mc} : memory cycle time component of memory cost = $371.5/mc^{0.55}$

C_{mb} : memory bank component of memory cost = $0.1875 \cdot mb$.

The division of cost between the CPU and memory is taken from [BEL71]. Most of the CPU cost functions are based on a rough division of cost among the various units of the actual 360/91. The cost division assumed was:

Instruction unit : 15%

Fixed Point unit : 25%

Floating Point unit : 45%

Main Storage Control : 15%

The cost function for the memory cycle time was derived from a curve fitted to memory speed, cost and size data for various System 360 Models, obtained from [BEL71]. The curve obtained was:

$$\text{Memory cost} = 32.9 + 370.0 \cdot \frac{(\text{memory size})^{0.64}}{(\text{cycle time})^{0.55}}$$

where cost is in K$,

size is in multiples of 256 K bytes and cycle time is in μsec.

A further assumption that the cycle time accounts for 95%, and the banking structure for 3%, of the memory cost in the 360/91, yields the above cost functions.

5.4 Traces Used in the Experiments

In the optimization experiments that were conducted on the case-study system, three program traces were used for generating the control streams to drive the low-level model simulator. These traces are sections of actual program traces obtained as described in Sec. 5.2.1. The traces are:

a) EIGEN: a program written in FORTRAN-G, to find the eigenvalues of a 14 x 14 matrix chosen from [GRE69]. It uses the subroutines TRED1 (to reduce the symmetric matrix to a tridiagonal one) and TOL1 (to determine the eigenvalues of the matrix). These two routines were taken from the Eigensystem Subroutine Package (EISPACK) of the National Activity to Test Software project. The section of the trace used, was the first four iterations of the TRED1 subroutine. A summary of the instruction mix of this section is given in Table 5.1. It can be seen that most of the conditional branches branch back into the instruction stream based on the value of a counter register, i.e., the program has a large number of instruction loops, with sizes ranging from 16-1024 bytes.

b) GAUSS: a program written in FORTRAN-G that used Gaussian elimination to solve a linear system of equations of order 20, taken from [GRE69]. It uses the subroutine GAUSZ, from the EISPACK library, to solve the system. The section of the trace used was the first four iterations of the forward elimination loop of GAUSZ. A summary of the instruction mix

Table 5.1 - Instruction Mix Summary of EIGEN

Total number of instructions: 14395

Percentage mix:

Fixed point instructions:	51.39
Address-to-register loads:	2.02
Register-to-register moves:	8.77
Memory-to-register loads:	9.07
Register-to-memory stores:	5.04
Computational instructions:	
On register operands:	17.51
On memory operands:	8.95
Floating point instructions:	40.41
Register-to-register moves:	0.14
Memory-to-register loads:	12.26
Register-to-memory stores:	5.81
Computational instructions:	
On register operands:	12.96
On memory operands:	9.24
Branches:	8.20
Unconditional branches:	0.11
Conditional branches:	
On the condition code:	1.21
On a counter register:	6.88
Taken:	6.54
Target back in the stream:	6.46
Mean distance of back-target (in bytes):	44.59

Histogram of back-target distance (in bytes):

Range	Percent
1: 2	0.00
2: 4	0.00
4: 8	0.00
8: 16	0.00
16: 32	10.43
32: 64	53.87
64: 128	26.24
128: 256	0.00
256: 512	4.52
512:1024	4.52
>=1024	0.43

of this section is given in Table 5.2. This program has about twice the percentages of branches in EIGEN. However, only approximately half of these are loop iteration branches.

c) ERROR: a scaled-down version of a FORTRAN program, that is used as a benchmark by the Computing Services Office of the University of Illinois. A summary of its instruction mix is given in Table 5.3. This program has a large amount of double precision floating point computation, done in predominantly straight-line code, i.e., there are very few branches.

5.5 Discussion of Optimization Experiments

We now discuss some of the optimization experiments conducted using the procedure described in Chapter 4, on the case-study system for the traces described in the last section.

5.5.1 An Iteration of the Global Optimization Procedure

Table 5.4 is an example of the results of a typical iteration of the global optimization procedure. As described in Sec. 4.4.6, the recalibration procedure chooses a recalibration set from the calibration set. This is indicated in the top half of the table. The distances indicated in the table are in terms of the grid metrics for the various dimensions. The recalibration set is used to calibrate the hyperplane model at the current reference system. The error in the model for the systems in the calibration set is printed here for illustrative purposes, but may actually be used to control the procedure. The coefficients of the model, along with the cost model parameters described in Sec. 5.3, are fed to the local optimization procedure, which determines the locally optimum system in the feasibility region demarcated as in Sec. 4.5.3. The movement rule of Sec. 4.4.4 is then used to determine the new reference system for the next

Table 5.2 - Instruction Mix Summay of GAUSS

Total number of instructions: 19380

Percentage mix:

Fixed point instructions:	38.36
Address-to-register loads:	0.75
Register-to-register moves:	5.61
Memory-to-register loads:	5.98
Register-to-memory stores:	2.98
Computational instructions:	
On register operands:	16.60
On memory operands:	6.43
Floating point instructions:	46.28
Register-to-register moves:	0.19
Memory-to-register loads:	13.90
Register-to-memory stores:	6.76
Computational instructions:	
On register operands:	14.77
On memory operands:	10.66
Branches:	15.35
Unconditional branches:	0.04
Conditional branches:	
On the condition code:	7.60
On a counter register:	7.72
Taken:	14.62
Target back in the stream:	7.67
Mean distance of back-target (in bytes):	44.46

Histogram of back-target distance (in bytes):

Range	Percent
1: 2	0.00
2: 4	0.00
4: 8	0.00
8: 16	1.28
16: 32	3.84
32: 64	28.13
64: 128	61.91
128: 256	1.28
256: 512	3.43
512:1024	0.00
>=1024	0.13

Table 5.3 - Instruction Mix Summay of ERROR

```
Total number of instructions:   13368

Percentage mix:

Fixed point instructions:       4.19
   Address-to-register loads:       0.41
   Register-to-register moves:      0.27
   Memory-to-register loads:        1.81
   Register-to-memory stores:       1.18
   Computational instructions:
      On register operands:         0.02
      On memory operands:           0.49

Floating point instructions:      93.78
   Register-to-register moves:       0.74
   Memory-to-register loads:        24.04
   Register-to-memory stores:       15.62
   Computational instructions:
      On register operands:         12.37
      On memory operands:           41.02

Branches:       2.02
   Unconditional branches:        0.89
   Conditional branches:
      On the condition code:        1.00
      On a counter register:        0.13
      Taken:        0.10

      Target back in the stream:        0.10
      Mean distance of back-target (in bytes):   2423.14
      Histogram of back-target distance (in bytes):
            Range  Percent
          1:    2     0.00
          2:    4     0.00
          4:    8     0.00
          8:   16     0.00
         16:   32     0.00
         32:   64     0.00
         64:  128     0.00
        128:  256    14.29
        256:  512     0.00
        512: 1024     0.00
         >=1024      85.71
```

Table 5.4 - An Iteration of the Global Optimization Procedure

Calibration set: 00 02 03 04 05 06 07 08 09 10 11 54
Loopmode: 1 (on)
Reference system: 54
mc grid metric: 6 CPU cycles

Distance of systems from reference system:

System	mc	mb	ib	fx	fl	lm	Norm^2
54	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	-1.00	-1.00	-1.00	1.00	0.00	4.00
5	1.00	0.00	-1.00	-1.00	1.00	0.00	4.00
6	1.00	-1.00	-1.00	0.00	1.00	0.00	4.00
9	1.00	-1.00	-1.00	-1.00	0.00	0.00	4.00
11	1.00	-1.00	0.00	-1.00	1.00	0.00	4.00
0	1.00	-1.00	-1.00	-1.00	1.00	0.00	5.00
7	1.00	-1.00	-1.00	1.00	1.00	0.00	5.00
8	1.00	-1.00	-1.00	-1.00	-1.00	0.00	5.00
3	2.00	-1.00	-1.00	-1.00	1.00	0.00	8.00
4	1.00	-2.00	-1.00	-1.00	1.00	0.00	8.00
10	1.00	-1.00	-2.00	-1.00	1.00	0.00	8.00

Recalibration set: 54 2 5 6 9 11 0 7 8

Hyperplane model coefficients:

Constant	mc	mb	ib	fx	fl
0.29811	-0.01217	0.01694	0.00180	0.00961	0.00894

Error in hyperplane model predictions of throughput:

System	Observed	Predicted	%Error
54	0.35112	0.35409	-0.85
2	0.33764	0.33467	0.88
5	0.28156	0.27858	1.06
6	0.26681	0.27126	-1.67
9	0.24924	0.25271	-1.39
11	0.26642	0.26344	1.12
0	0.25669	0.26164	-1.93
7	0.28458	0.28087	1.30
8	0.24699	0.24377	1.30
3	0.20186	0.18862	6.56
4	0.22391	0.24470	-9.29
10	0.19933	0.25984	-30.36

	mc	mb	ib	fx	fl
Initial reference system	6.	32.	16.	2.	2.
Optimum system	7.42	98.04	11.77	2.72	2.54
New reference system	8.	64.	8.	3.	3.

Cost/performance value of the optimum system: 323.3087
Throughput of the optimum system: 0.3752
Memory,CPU and system costs: 61.83 59.47 121.30
Relative memory parameter costs: 123.41 20.38
Relative CPU parameter costs: 10.47 31.42 42.43 5.00 15.00

iteration of the global procedure. Not shown is the application of the stopping rule, or the possible run of the low-level model for the new reference system, for future calibration. The cost and cost/performance values of the locally optimum system are shown purely for illustrative purposes. The relative memory parameter costs are the costs of memory cycle time and memory banks, respectively. The relative CPU parameter costs are the costs of the instruction unit (without loop-mode), the fixed point unit, the floating point unit, loop-mode and the fixed CPU costs respectively.

In the following sections, we use the following terminology:

1) Convergence sequence: The sequence of reference systems generated by the global optimization procedure, before oscillation occurred.

2) Sensitivity sequence: The sequence of systems examined, after convergence, by the individual parameter sensitivity analysis procedure described in Sec. 4.4.8.2.

3) Sensitivity report: A summary of the sensitivity sequence, listing the sensitivity of cost, performance and cost/performance to the various system parameters at the optimum system.

The sensitivity sequence will be shown for only one experiment, since the others are very similar and, consequently not very informative.

5.5.2 Experiments on EIGEN

The EIGEN program was run on various system configurations for experiments 1 and 2.

5.5.2.1 Optimization on the $\ell m = 1$ hyperplane

Since EIGEN was the first trace that the experiments were tried on, the optimization was run for three different starting points on the $\ell m = 1$ hyperplane, in an attempt to establish confidence in its convergence.

5.5.2.1.1 Experiment 1a:

Table 5.5 lists the convergence sequence with the starting point at the normal system (parameters: 12, 16, 8, 1, 3) on the $\ell m = 1$ hyperplane. Notice that the mere repetition of a calibration point as a reference system does not indicate oscillation. Thus the repetition of system 18 on iterations 11 and 13 does not constitute an oscillation, because iteration 12 introduces another reference system, 60, for possible inclusion in future calibration sets, which may change the analytical model based at system 18. That this does happen is shown by the fact that the model in iteration 14 yields a new reference system, 61. Thus a pair of reference systems must be repeated twice in succession for an oscillation.

Table 5.6 lists the sensitivity sequence for this experiment. The sequence starts at the better of the two systems involved in the oscillation that ended the convergence sequence. Parameters are perturbed individually and the base system is changed if a decrease in cost/performance is achieved. The procedure is continued until a pass through all the parameters causes no change to the base system.

The optimum system reached is (6, 32, 8, 3, 3). It is interesting to note that the optimization procedure reaches an oscillation hypercube containing the optimum value for all the system parameters except $f\ell$. Table 5.7 which lists the sensitivity report for this experiment, shows that the projections of both the performance and cost surface (and consequently the cost/performance surface) onto the $f\ell$ coordinate hyperplane are quite flat in the region $f\ell$ = 2 to 3. We believe that this slope is within the regression error bounds of the procedure, thus causing the optimization error. We discuss this further in Sec. 6.3.1.

Table 5.5 - Convergence Sequence for Experiment 1a on EIGEN

Iteration	Reference System	Reference System Parameters				
		mc	mb	ib	fx	fl
0	0	12	16	8	1	3
1	54	6	32	16	2	2
2	55	8	64	8	3	3
3	98	7	128	16	3	3
4	99	5	64	8	2	3
5	100	9	128	16	3	3
6	101	10	256	8	3	2
7	58	6	128	16	3	1
8	28	5	64	8	3	2
9	36	8	128	16	3	1
10	102	9	64	8	3	2
11	18	6	32	16	3	1
12	60	8	64	8	3	2
13	18	6	32	16	3	1
14	61	8	16	8	3	2
15	18	6	32	16	3	1
16	15	7	64	8	3	2
17	18	6	32	16	3	1
18	15	7	64	8	3	2
Oscillation Hypercube		6:7	32:64	8:16	3	1:2

Table 5.6 - Sensitivity Sequence for Experiment 1a on EIGEN

	System	System Parameters					Cost/Performance
		mc	mb	ib	fx	fl	
Base -->	15	7	64	8	3	2	303.29
	63	6	64	8	3	2	299.98
	60	8	64	8	3	2	309.23
	28	5	64	8	3	2	303.63
Base -->	63	6	64	8	3	2	299.98
	72	6	32	8	3	2	296.21
	73	6	128	8	3	2	311.54
	74	6	16	8	3	2	310.75
Base -->	72	6	32	8	3	2	296.21
	75	6	32	4	3	2	368.07
	76	6	32	16	3	2	313.45
	77	6	32	8	2	2	327.19
	78	6	32	8	3	1	297.33
	79	6	32	8	3	3	291.72
Base -->	79	6	32	8	3	3	291.72
	End of Pass 1						
Base -->	79	6	32	8	3	3	291.72
	88	5	32	8	3	3	294.19
	89	7	32	8	3	3	295.69
	95	6	16	8	3	3	306.26
	108	6	64	8	3	3	294.93
	96	6	32	4	3	3	369.34
	27	6	32	16	3	3	309.06
	97	6	32	8	2	3	323.46
	72	6	32	8	3	2	296.21
	End of Pass 2						
	Optimum System	6	32	8	3	3	

Table 5.7 - Sensitivity Report for Experiments 1a and 1b on EIGEN

Optimum System : (6,32,8,3,3)

System	System Parameters					Performance		Cost		Cost/Performance	
	mc	mb	ib	fx	fl	Actual	%change from optimum	Actual	%change from optimum	Actual	%change from optimum
88	5	32	8	3	3	0.4450	+4.2	130.92	+4.8	294.19	+0.9
79	6	32	8	3	3	0.4272	0.0	124.63	0.0	291.72	0.0
89	7	32	8	3	3	0.4051	-5.2	119.78	-3.9	295.69	+1.4
95	6	16	8	3	3	0.4027	-5.7	123.34	-1.0	306.26	+5.0
79	6	32	8	3	3	0.4272	0.0	124.63	0.0	291.72	0.0
108	6	64	8	3	3	0.4313	+1.0	127.21	+2.1	294.93	+1.1
153	6	128	8	3	3	0.4329	+1.3	132.37	+6.2	305.79	+4.8
96	6	32	4	3	3	0.3367	-21.2	124.34	-0.2	369.34	+26.8
79	6	32	8	3	3	0.4272	0.0	124.63	0.0	291.72	0.0
27	6	32	16	3	3	0.4051	-5.2	125.20	+0.5	309.06	+5.9
97	6	32	8	2	3	0.3765	-11.9	121.78	-2.3	323.46	+10.9
79	6	32	8	3	3	0.4272	0.0	124.63	0.0	291.72	0.0
78	6	32	8	3	1	0.4038	-5.5	120.07	-3.7	297.33	+1.9
72	6	32	8	3	2	0.4111	-3.8	121.78	-2.3	296.21	+1.5
79	6	32	8	3	3	0.4272	0.0	124.63	0.0	291.72	0.0

5.5.2.1.2 Experiment 1b

In this experiment, the procedure was purposely started at a point very far away from the optimum reached by experiment 1a - viz. (18, 8, 4, 1, 1). Table 5.8 lists the convergence sequence for this experiment which eventually reaches the optimum system (6, 32, 8, 3, 3). In this experiment, the procedure reaches an oscillation hypercube containing the optimum value for all the system parameters except mb. The sensitivity report in Table 5.7 (the same as for experiment 1a), indicates that the projections of the cost, performance, and hence cost/performance, surfaces onto the mb coordinate hyperplane are also very flat in the region mb = 32 to 64. We believe this, too, to be within the regression error bounds of the procedure.

5.5.2.1.3 Experiment 1c

In this experiment, the procedure was started at yet another point on the $\ell m = 1$ hyperplane (18, 16, 8, 1, 3). This is one of the points in the initial calibration set. Table 5.9 lists the convergence sequence for this experiment. It will be observed that the optimum hypercube reached is substantially different from experiments 1a and 1b, in the mc dimension, 10:11 against 6:7 and 5:6. The reason for this becomes clear from looking at the sensitivity report for this experiment in Table 5.10. The system (10, 32, 8, 3, 3) is seen to be a locally optimum system, on the evidence of the rough analysis conducted by the individual parameter sensitivity procedure. Figure 5.1 is a plot of the projections of the performance and cost/performance surfaces onto the mc coordinate hyperplane, with the other parameters fixed at (32, 8, 3, 3) respectively. The plot clearly shows the anomalous behavior of the cost/performance surface, along the mc dimension that led the procedure to find a local optimum in this experiment.

Table 5.8 - Convergence Sequence for Experiment 1b on EIGEN

Iteration	Reference System	Reference System Parameters				
		mc	mb	ib	fx	fl
0	32	18	8	4	1	1
1	103	21	16	8	1	1
2	104	24	32	16	2	1
3	51	18	64	32	3	1
4	105	13	128	16	2	2
5	106	15	64	32	1	3
6	107	10	32	16	2	2
7	108	6	64	8	3	3
8	109	8	32	16	3	3
9	108	6	64	8	3	3
10	110	8	128	16	3	3
11	108	6	64	8	3	3
12	111	8	32	16	2	2
13	112	6	64	32	3	1
14	111	8	32	16	2	2
15	113	7	16	8	2	3
16	27	6	32	16	3	3
17	13	7	64	8	3	3
18	69	6	128	16	3	3
19	28	5	64	8	3	2
20	96	6	32	4	3	3
21	114	5	64	8	2	2
22	115	4	128	16	3	3
23	28	5	64	8	3	2
24	69	6	128	16	3	3
25	28	5	64	8	3	2
26	69	6	128	16	3	3
Oscillation Hypercube		5:6	64:128	8:16	3	2:3

Table 5.9 - Convergence Sequence of Experiment 1c on EIGEN

Iteration	Reference System	Reference System Parameters				
		mc	mb	ib	fx	fl
0	3	18	16	8	1	3
1	116	12	32	16	2	2
2	117	7	64	32	3	3
3	12	9	32	16	2	2
4	13	7	64	8	3	3
5	56	6	128	16	3	3
6	118	10	64	8	2	3
7	56	6	128	16	3	2
8	119	10	64	8	3	3
9	120	6	128	4	3	2
10	121	9	64	8	3	3
11	122	11	32	16	3	3
12	123	14	64	8	3	2
13	124	12	128	16	3	3
14	125	13	64	8	3	3
15	126	11	128	16	3	2
16	127	10	64	8	3	1
17	128	9	128	4	3	2
18	129	10	64	8	2	1
19	130	11	32	16	3	2
20	131	10	64	32	3	1
21	132	11	32	16	3	1
22	133	10	64	8	3	2
23	134	11	32	16	2	1
24	135	12	16	8	3	1
25	136	13	32	16	3	2
26	137	12	64	32	3	1
27	138	13	32	16	3	1
28	139	12	16	32	3	2
29	134	11	32	16	3	1
30	133	10	64	8	3	2
31	134	11	32	16	3	1
32	133	10	64	8	3	2
Oscillation Hypercube		10:11	32:64	8:16	3	1:2

Table 5.10 - Convergence Sequence for Experiment 1c on EIGEN

Optimum System : (10,32,8,3,3)

System	System Parameters					Performance		Cost		Cost/Performance	
	mc	mb	ib	fx	fl	Actual	%change from optimum	Actual	%change from optimum	Actual	%change from optimum
143	9	32	8	3	3	0.3625	+1.1	112.71	+2.5	310.93	+1.4
141	10	32	8	3	3	0.3586	0.0	110.02	0.0	306.78	0.0
144	11	32	8	3	3	0.3396	-5.3	107.72	-2.1	317.24	+3.4
146	10	16	8	3	3	0.3205	-10.6	108.73	-1.2	339.30	+10.6
141	10	32	8	3	3	0.3586	0.0	110.02	0.0	306.78	0.0
119	10	64	8	3	3	0.3656	+2.0	112.60	+2.4	307.96	+0.4
147	10	32	4	3	3	0.2569	-28.4	109.74	-0.25	427.16	+39.2
141	10	32	8	3	3	0.3586	0.0	110.02	0.0	306.78	0.0
148	10	32	16	3	3	0.3566	-0.6	110.59	+0.5	310.15	+1.1
149	10	32	8	2	3	0.3247	-9.5	107.17	-2.6	330.06	+7.6
141	10	32	8	3	3	0.3586	0.0	110.02	0.0	306.78	0.0
145	10	32	8	3	2	0.3446	-3.9	107.17	-2.6	311.00	+1.4
141	10	32	8	3	3	0.3586	0.0	110.02	0.0	306.78	0.0

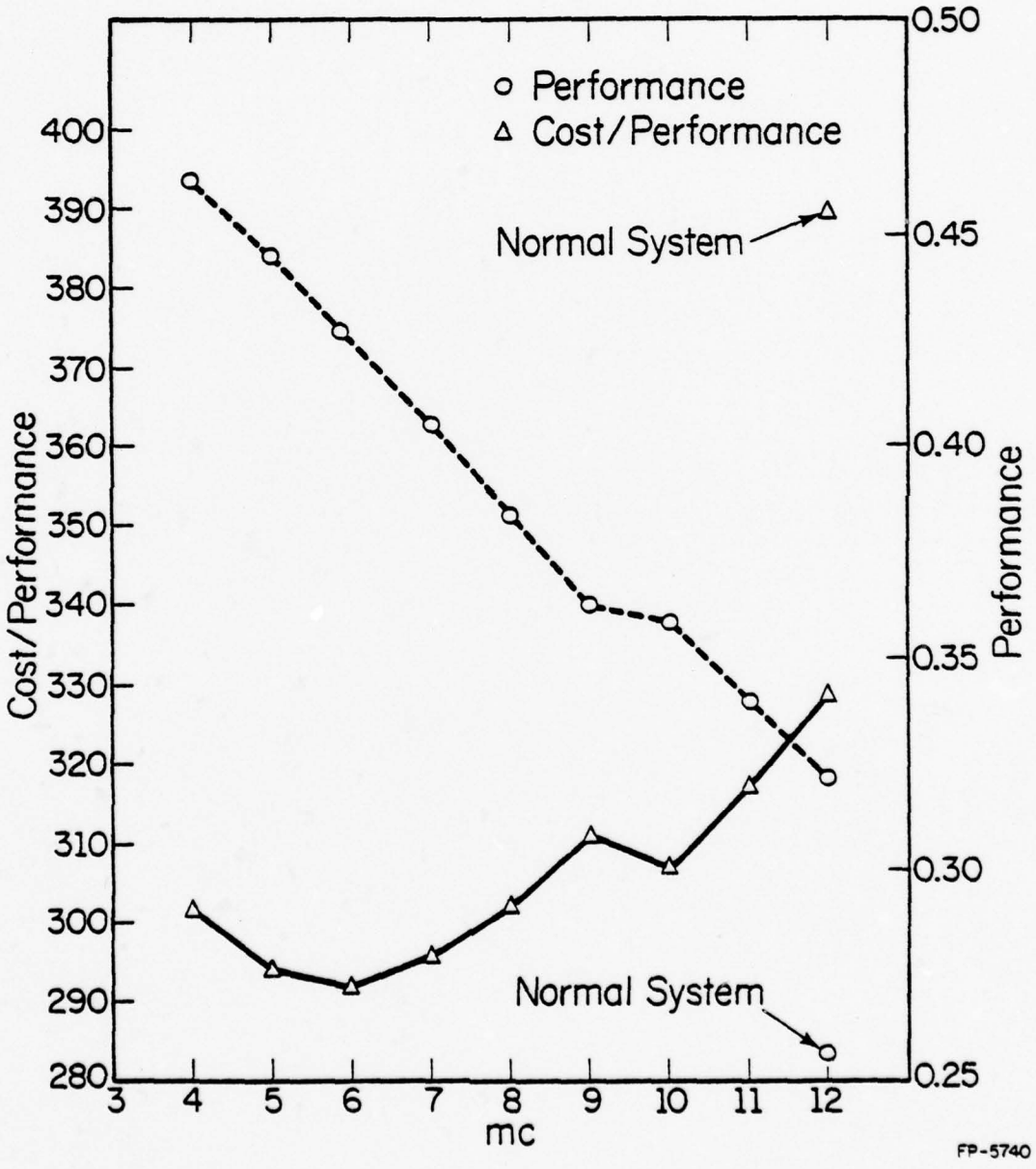


Figure 5.1 Projections of performance and cost/performance surfaces onto the mc coordinate hyperplane for EIGEN.

We believe that the main reason for the procedure arriving at this secondary local optimum, is the shortening of the mc grid metric just when the procedure happened to be exploring the region of the secondary (local) optimum. The purpose of gradually shortening the mc grid metric was to force the procedure to maintain a global perspective during the initial stages, when it has a very small calibration set, but to increasingly localize its perspective as the possible choices for the recalibration set increase. For experiment 1a and 1b, this shortening happened when the procedure had reached the region of the global optimum of Figure 5.1. For experiment 1c however, some regression error resulted in the procedure being nearer the local optimum, when the mc grid metric was being shortened. This can be seen by comparing the mc values of the reference system on iteration 12 of the three experiments - they are 8, 8 and 14 respectively. Localizing the perspective then rendered the procedure incapable of looking beyond the region of the local optimum.

The performance and cost/performance of the normal system are also indicated on Figure 5.1 to illustrate the flatness of the cost/performance surface near the optimum along the mc dimension.

Here again, the error of experiment 1 along the fℓ dimension has re-occurred, and for the same reason.

5.5.2.2 Experiment 2: Optimization on the ℓm = 0 hyperplane

This experiment was conducted by running EIGEN on systems that had the loop-mode feature turned off (ℓm = 0). The starting point was the otherwise normal system (12, 16, 8, 1, 3). Table 5.11 lists the convergence sequence for this experiment and Table 5.12 is the sensitivity report at the actual optimum system (6, 32, 8, 3, 3). The oscillation hypercube reached by the procedure does not contain the optimum value for

Table 5.11 - Convergence Sequence for Experiment 2 on EIGEN

Iteration	Reference System	Reference System Parameters				
		mc	mb	ib	fx	fl
0	1	12	16	8	1	3
1	38	9	32	16	2	2
2	39	6	64	8	1	1
3	40	7	128	16	1	2
4	41	4	64	8	2	3
5	42	6	32	4	3	2
6	43	5	16	8	2	1
7	44	6	32	16	3	1
8	45	5	16	32	2	2
9	46	4	32	16	1	1
10	43	5	16	8	2	1
11	47	6	32	4	3	1
12	48	5	64	2	2	2
13	49	6	32	4	3	3
14	57	7	16	2	2	2
15	60	8	32	4	3	1
16	61	9	64	8	2	2
17	62	8	128	16	3	1
18	63	9	64	32	3	1
19	12	8	32	16	2	2
20	64	7	64	8	3	3
21	65	8	32	16	3	3
22	64	7	64	8	3	3
23	66	8	128	16	3	3
24	67	9	64	8	3	2
25	66	8	128	16	3	3
26	67	9	64	8	3	2
Oscillation Hypercube		8:9	64:128	8:16	3	2:3

Table 5.12 - Sensitivity Report for Experiment 2 on EIGEN

Optimum System : (6,32,8,3,3)

System	System Parameters					Performance		Cost		Cost/Performance	
	mc	mb	ib	fx	fl	Actual	%change from optimum	Actual	%change from optimum	Actual	%change from optimum
80	5	32	8	3	3	0.4178	+4.5	128.07	+5.2	306.52	+0.6
79	6	32	8	3	3	0.3998	0.0	121.78	0.0	304.58	0.0
81	7	32	8	3	3	0.3735	-6.6	116.93	-4.0	313.07	+2.8
86	8	32	8	3	3	0.3468	-13.3	113.05	-7.2	315.59	+3.6
82	6	16	8	3	3	0.3740	-6.5	120.49	-1.1	322.19	+5.8
79	6	32	8	3	3	0.3998	0.0	121.78	0.0	304.58	0.0
83	6	64	8	3	3	0.4066	+1.7	124.36	+2.1	305.88	+0.4
87	6	128	8	3	3	0.4104	+2.7	129.52	+6.4	315.59	+3.6
49	6	32	4	3	3	0.3298	-17.5	121.49	-0.2	368.43	+21.0
79	6	32	8	3	3	0.3998	0.0	121.78	0.0	304.58	0.0
84	6	32	16	3	3	0.3357	-16.0	122.35	+0.5	364.41	+19.6
85	6	32	8	2	3	0.3544	-11.1	118.93	-2.3	335.61	+10.2
79	6	32	8	3	3	0.3998	0.0	121.78	0.0	304.58	0.0
78	6	32	8	3	1	0.3821	-4.4	117.22	-3.7	306.79	+0.7
73	6	32	8	3	2	0.3869	-3.2	118.93	-2.3	307.39	+0.9
79	6	32	8	3	3	0.3998	0.0	121.78	0.0	304.58	0.0

two-dimensions - mc and mb. In both cases, the flatness of the projections of the cost/performance surface onto the coordinate hyperplanes are seen to be quite small, and are possibly within the regression error bounds of the procedure.

5.5.2.3 Analysis of the optimum architecture for EIGEN

Experiments 1 and 2 show that the optimum system for EIGEN is (6, 32, 8, 3, 3) on the $\ell m = 1$ hyperplane. We now analyze this architecture in more detail.

5.5.2.3.1 Orthogonality of the cost/performance surface

The shape of the cost/performance surface seems to be fairly orthogonal with respect to its system parameters. This is illustrated by the fact that the optimum system is (6, 32, 8, 3, 3) on both the ℓm hyperplanes. Further, the local optimum reached in experiment 1c, was different in the mc dimension (10 as opposed to 6). But this did not affect the optimum values for the other parameters, viz (32, 8, 3, 3). This seems to indicate that there is very little interaction between the parameters near the optimum on the cost/performance surface.

5.5.2.3.2 Instruction unit issues

Loop-mode is indeed cost effective by to a surprisingly small extent. Thus in the optimum system (6, 32, 8, 3, 3), removing loop-mode causes a performance and a cost/performance degradation of only 6.4% and 4.4%. We expect that this difference would be even smaller, if a cache, which is a cost-effective technique of achieving an even lower memory cycle time, were used. The small contribution that loop-mode makes to performance also explains why the optimum system on both the ℓm hyperplanes had the same values for the other system parameters, viz (6, 32, 8, 3, 3).

In particular, it is interesting that the optimum ib value is 8 on both the ℓm hyperplanes. The choice of the most cost-effective value of ib involves a tradeoff between three factors. The necessity to maintain a degree of look-ahead sufficient to ensure a high instruction supply bandwidth, demands a high value of ib. So does the possibility of holding increasingly larger instruction loops in the buffer with loop-mode However, the occurrence of branches renders a number of prefetched instructions superfluous. Since the fetching of these instructions uses critical resources such as memory banks, the high occurrence of superfluity argues for a low degree of prefetch, i.e., a low value for ib. We will illustrate this tradeoff on EIGEN.

With $\ell m = 0$, the buffer is used solely to hold prefetched instructions. The optimum value arrived at for ib is 8. Table 5.12 indicates that reducing ib to 4 degrades performance by as much as 17.5%, because it drastically reduces the instruction supply bandwidth. However, increasing ib to 16, causes a comparable degradation in performance (16%) due to the increase in superfluity. Thus ib = 8 represents the choice that best trades-off these two factors.

With $\ell m = 1$, loop-mode enters the tradeoff considerations. But here again, the optimum choice for ib is 8. Table 5.7 shows that if ib = 4, the buffer is neither large enough for an adequate instruction supply bandwidth in non-looping situations, nor is it large enough to hold any loops (the buffer can hold 4* ib bytes of instructions - see Table 5.1) - hence the performance degradation of 21.2%. However, increasing ib to 16 also causes a performance degradation of 5.2%. This is despite the fact that 64.3% (see Table 5.1) of all the looping branches in EIGEN have a

target distance of less than 64 bytes (against 10.4% for 32 bytes). The degradation due to superfluity is evidently greater than the increased bandwidth due to loop-mode. This suggests that loop-mode is really cost-effective only for small loops, where the branch decision and target fetching time forms a large percentage of the loop execution time.

5.5.2.3.3 Execution unit issues

A high bandwidth fixed point unit is vital for performance. This is indicated in Table 5.7, by the fact that reducing fx from 3 to 2 causes an 11.9% degradation in performance and a 10.9% degradation in cost/performance. On the other hand, fℓ has a much smaller effect on system performance and cost/performance. Thus reducing fℓ from 3 to 2 causes only a 3.8% degradation in performance and a 1.5% degradation in cost/performance. This can also be seen from the convergence sequences, where fx stays fairly steady at 3, while fℓ moves unpredictably over the range 1 to 3.

We believe that this is linked with the fact that the proportion of fixed to floating point instructions in EIGEN is 1.3:1. We will contrast this with GAUSS in Sec. 5.5.3.3.

5.5.2.3.4 Memory issues

Table 5.7 shows that the latency of the memory has a greater influence on performance than its bandwidth. Thus increasing the number of banks from 32 to 64 (128), causes a mere 1.0% (1.3%) improvement in performance. This suggests that the number of memory conflicts has not decreased substantially upon increasing mb. However reducing mc from 6 to 5, causes a performance improvement of 4.2%, which is substantially more than the effect of mb.

5.5.3 Experiments on GAUSS

Experiments 3 and 4 deal with the GAUSS program run on various system configurations.

5.5.3.1 Experiment 3: Optimization on the $\ell m = 1$ hyperplane

Table 5.13 lists the convergence sequence for this experiment, with the starting point at the normal system (12, 16, 18, 1, 3) on the $\ell m = 1$ hyperplane. The procedure converges fairly rapidly compared with the EIGEN experiments. It is also interesting to observe that the procedure examines a much larger range (8 to 128) along the ib dimension than it did for EIGEN (8 to 32). This is because of the greater effect of loop-mode which we discuss in Sec. 5.5.3.3.

For this experiment Table 5.14 lists the sensitivity report about the final optimum (4, 16, 16, 3, 3). Thus the oscillation hypercube reached by the procedure does not contain the optimum value for two-dimensions - mc and fx. Here again, the flatness of the cost/performance surface projections onto the coordinate hyperplanes possibly explains the errors - since the slopes seem to be within the regression error bounds of the procedure.

5.5.3.2 Experiment 4: Optimization on the $\ell m = 0$ hyperplane

In this experiment, conducted using systems on the $\ell m = 0$ hyperplane, the otherwise normal system (12, 16, 8, 1, 3) was again used as the starting point. As in the previous experiment, the procedure converged fairly rapidly. The convergence sequence is listed in Table 5.15. In direct contrast to experiment 3, the ib range examined was very small and in the opposite direction from experiment 3 (2 to 8).

For this experiment, Table 5.16 lists the sensitivity report about the optimum system (4, 32, 4, 1, 3). The oscillation hyper-

Table 5.13 - Convergence Sequence for Experiment 3 on GAUSS

Iteration	Reference System	Reference System Parameters				
		mc	mb	ib	fx	fl
0	0	12	16	8	1	3
1	12	13	32	16	1	2
2	13	8	16	32	1	3
3	14	6	32	64	1	2
4	15	7	16	128	1	3
5	16	9	32	64	1	2
6	17	8	64	128	1	3
7	14	6	32	64	1	2
8	15	7	16	128	1	3
9	18	5	32	128	1	2
10	19	6	16	64	1	3
11	20	7	32	32	1	3
12	19	6	16	64	1	3
13	21	5	32	32	1	3
14	22	6	16	16	1	3
15	23	5	32	8	1	3
16	22	6	16	16	1	3
17	21	5	32	32	1	3
18	22	6	16	16	1	3
19	21	5	32	32	1	3
Oscillation Hypercube		5:6	16:32	16:32	1	3

Table.14 - Sensitivity Report for Experiment 3 on GAUSS

Optimum System : (4,16,16,3,3)

System	System Parameters					Performance		Cost		Cost/Performance	
	mc	mb	ib	fx	fl	Actual	%change from optimum	Actual	%change from optimum	Actual	%change from optimum
42	3	16	16	3	3	0.2868	0.0	151.58	+9.2	528.60	+9.2
41	4	16	16	3	3	0.2868	0.0	138.80	0.0	484.05	0.0
33	5	16	16	3	3	0.2652	-7.5	130.20	-6.2	490.88	+1.0
49	6	16	16	3	3	0.2461	-14.2	123.91	-10.7	503.40	+4.0
43	4	8	16	3	3	0.2812	-2.0	138.16	-0.5	491.32	+1.5
41	4	16	16	3	3	0.2868	0.0	138.80	0.0	484.05	0.0
44	4	32	16	3	3	0.2885	+0.6	140.09	+0.9	485.56	+0.3
45	4	16	8	3	3	0.2350	-18.1	138.23	-0.4	588.34	+21.6
41	4	16	16	3	3	0.2868	0.0	138.80	0.0	484.05	0.0
46	4	16	32	3	3	0.2746	-4.3	139.94	+0.8	509.65	+5.3
47	4	16	16	1	3	0.2663	-7.2	134.24	-3.3	504.10	+4.1
39	4	16	16	2	3	0.2806	-2.2	135.95	-2.1	484.49	+0.1
41	4	16	16	3	3	0.2868	0.0	138.80	0.0	484.05	0.0
48	4	16	16	3	2	0.2537	-11.5	135.95	-2.1	535.95	+10.7
41	4	16	16	3	3	0.2868	0.0	138.80	0.0	484.05	0.0

Table 5.15 - Convergence Sequence for Experiment 4 on GAUSS

Iteration	Reference System	Reference System Parameters				
		mc	mb	ib	fx	fl
0	1	12	16	8	1	3
1	12	9	32	4	1	2
2	13	6	16	8	1	1
3	14	5	32	4	2	2
4	15	7	64	8	1	3
5	16	5	32	4	1	3
6	17	6	64	8	1	3
7	16	5	32	4	1	3
8	18	4	64	2	1	3
9	19	6	32	4	1	2
10	20	5	64	8	1	3
11	21	6	32	4	1	3
12	20	5	64	8	1	3
13	21	6	32	4	1	3
Oscillation Hypercube		5:6	32:64	4:8	1	3

Table 5.16 - Sensitivity Report for Experiment 4 on GAUSS

Optimum System : (4,32,4,1,3)

System	System Parameters					Performance		Cost		Cost/Performance	
	mc	mb	ib	fx	fl	Actual	%change from optimum	Actual	%change from optimum	Actual	%change from optimum
35	3	32	4	1	3	0.2296	0.0	144.60	+9.7	629.82	+9.7
32	4	32	4	1	3	0.2296	0.0	131.83	0.0	574.17	0.0
16	5	32	4	1	3	0.2078	-9.5	123.22	-6.5	592.98	+3.3
21	6	32	4	1	3	0.1872	-18.5	116.93	-11.3	624.59	+8.8
37	4	16	4	1	3	0.2265	-1.4	130.54	-1.0	576.29	+0.4
32	4	32	4	1	3	0.2296	0.0	131.83	0.0	574.17	0.0
38	4	64	4	1	3	0.2306	+0.4	134.41	+2.0	582.89	+1.5
34	4	32	2	1	3	0.2109	-8.1	131.69	-0.1	624.41	+8.8
32	4	32	4	1	3	0.2296	0.0	131.83	0.0	574.17	0.0
28	4	32	8	1	3	0.2220	-3.3	132.11	+0.2	595.16	+3.7
32	4	32	4	1	3	0.2296	0.0	131.83	0.0	574.17	0.0
39	4	32	4	2	3	0.2299	+0.1	133.54	+1.3	580.79	+1.2
41	4	32	4	3	3	0.2327	+1.4	136.39	+3.5	586.03	+2.1
40	4	32	4	1	2	0.2072	-9.8	128.98	-2.2	622.42	+8.4
32	4	32	4	1	3	0.2296	0.0	131.83	0.0	574.17	0.0

cube thus does not contain the optimum value of the mc parameter for the same reason as the one given in experiment 3.

5.5.3.3 Analysis of the optimum architecture for GAUSS

Experiments 3 and 4 indicate that the optimum system for GAUSS is (4, 16, 16, 3, 3) on the $\ell m = 1$ hyperplane. We now analyze the results of these experiments in greater detail.

5.5.3.3.1 Non-orthogonal nature of the cost/performance surface for GAUSS

The results of experiments 3 and 4 indicate that the cost/performance surface departs much further from orthogonality with reference to its system parameters for GAUSS than for EIGEN. This can be seen from the different optima reached for the 2 ℓm hyperplanes. Thus:

a) With the increase in the instruction supply bandwidth due to the addition of loop-mode ($\ell m = 0$ to $\ell m = 1$), it becomes cost-effective to have a high bandwidth fixed point unit (fx = 1 to fx = 3). Further, with the demands made on memory for instruction fetching being reduced, the memory bandwidth can be reduced (mb = 32 to mb = 16).

b) The interaction between ℓm and ib is also clearly brought out and will be discussed further.

c) The sensitivity sequence for experiment 3 revealed that, while at a lower memory bandwidth (mc = 5), it is more cost-effective to have a low bandwidth fixed point unit (fx = 1) than a high bandwidth unit (fx = 3), the reverse is true when the memory bandwidth is increased (mc = 4). This can be seen from the cost/performance figures listed in Table 5.17.

Table 5.17 - Interaction Between mc and fx on the Cost/Performance Surface for GAUSS on the $\ell m = 1$ Hyperplane

System	System Parameters					Cost/Performance
	mc	mb	ib	fx	fl	
24	5	16	16	1	3	502.37
26	4	16	16	1	3	504.10
33	5	16	16	3	3	490.88
41	4	16	16	3	3	484.05

5.5.3.3.2 Instruction unit issues

Loop-mode has a much greater impact on system performance for GAUSS than for EIGEN. Thus, the optimum system on the $\ell m = 0$ hyperplane is 20% worse in performance and 18.6% worse in cost/performance than the optimum system on the $\ell m = 1$ hyperplane. This is also evident in the different optima reached on the two hyperplanes.

The instruction buffer tradeoff discussed earlier is brought out with great clarity for GAUSS. With $\ell m = 0$, ib = 4 is found to be the best tradeoff between prefetching and superfluity. The smaller degree of prefetch for GAUSS than for EIGEN, 4 against 8, is clearly related to the higher percentage of branches in the former (15.35% against 8.20%). This greatly increases the superfluity effect, forcing a low degree of prefetch. With $\ell m = 1$, ib = 16 is the best tradeoff between prefetching and loop-mode on the one hand and superfluity on the other. This is despite the fact that 95.2% of all the looping branches in GAUSS have their target distances less than 128 bytes (corresponding to ib = 32), against 33.3% for 64 bytes (ib = 16). This again illustrates the fact that loop-mode is cost-effective only for small loops. On the other hand ib = 8 is not sufficient, since only 5.1% of the looping branches have their target distances less than 32 bytes.

5.5.3.3.3 Execution unit issues

The relative importance of the two execution units is reversed in GAUSS, with respect to EIGEN, with the floating point unit gaining in prominence. This is seen from the convergence sequence of Table 5.15, where fx and fℓ stay steadily at 1 and 3 respectively. Further, the sensitivity report shows that decreasing fℓ from 3 to 2 causes performance

and cost/performance to degrade by 11.5% and 10.7%, against 2.1% and 0.1% for a corresponding change in fx.

Exactly the same observation can be made as for EIGEN - this relative importance is linked to the proportion of the two types of instructions in the program. For GAUSS, the proportion of fixed to floating point instructions is 0.83:1 against 1.3:1 for EIGEN.

5.5.3.3.4 Memory issues

A curious phenomenon occurs with GAUSS - viz. there is absolutely no performance increase to be had by decreasing mc from 4 to 3. Even increasing mb from 16 to 32, produces a marginal increase in performance of 0.6%. This suggests strongly that the bottleneck has shifted to system areas other than the memory for this program on the optimum system.

5.5.4 Experiments on ERROR

For experiments 5 and 6, the ERROR program was run on various system configurations.

5.5.4.1 Experiment 5: Optimization on the $\ell m = 1$ hyperplane

This experiment was conducted using systems on the $\ell m = 1$ hyperplane with the procedure started at the normal system (12, 16, 8, 1, 3). Table 5.18 lists the convergence sequence for the experiment, and Table 5.19 the sensitivity report around the optimum system (7, 64, 16, 1, 3). The oscillation hypercube reached a value of mc (14:15) which is very far away from the actual optimum value of 7. Examination of the cost/performance surface reveals that it is very flat over a wide range of values for the two memory parameters - mc and mb. This is clearly indicated in Table 5.20, which lists the cost/performance values for a number of

Table 5.18 - Convergence Sequence for Experiment 5 on ERROR

Iteration	Reference System	Reference System Parameters				
		mc	mb	ib	fx	fl
0	0	12	16	8	1	3
1	12	13	32	16	1	3
2	18	14	64	32	1	3
3	29	16	128	64	1	3
4	30	17	256	32	1	3
5	31	16	128	16	1	3
6	32	17	256	8	1	3
7	31	16	128	16	1	3
8	23	15	64	32	1	3
9	33	14	32	16	1	3
10	23	15	64	32	1	3
11	33	14	32	16	1	3
Oscillation Hypercube		14:15	32:64	16:32	1	3

Table 5.19 - Sensitivity Report for Experiment 5 on ERROR

Optimum System : (7,64,16,1,3)

System	System Parameters					Performance		Cost		Cost/Performance	
	mc	mb	ib	fx	fl	Actual	%change from optimum	Actual	%change from optimum	Actual	%change from optimum
45	6	64	16	1	3	0.6785	+3.4	123.22	+4.1	181.60	+0.7
44	7	64	16	1	3	0.6561	0.0	118.37	0.0	180.42	0.0
43	8	64	16	1	3	0.6277	-4.3	114.49	-3.3	182.40	+1.1
47	7	32	16	1	3	0.6245	-4.8	115.79	-2.2	185.41	+2.8
44	7	64	16	1	3	0.6561	0.0	118.37	0.0	180.42	0.0
48	7	128	16	1	3	0.6692	+2.0	123.53	+4.4	184.59	+2.3
49	7	64	8	1	3	0.6333	-3.5	117.80	-0.5	186.00	+3.1
44	7	64	16	1	3	0.6561	0.0	118.37	0.0	180.42	0.0
50	7	64	32	1	3	0.6208	-5.4	119.51	+1.0	192.50	+6.7
44	7	64	16	1	3	0.6561	0.0	118.37	0.0	180.42	0.0
51	7	64	16	2	3	0.6561	0.0	120.08	+1.4	183.03	+1.5
52	7	64	16	1	2	0.2383	-63.7	115.52	-2.4	484.83	+168.7
44	7	64	16	1	3	0.6561	0.0	118.37	0.0	180.42	0.0

Table 5.20 - Cost/Performance Figures for Some Systems on ERROR

Optimum System : (7,64,16,1,3)

System	System Parameters					Performance		Cost/Performance	
	mc	mb	ib	fx	fl	Actual	%change from optimum	Actual	%change from optimum
57	6	32	16	1	3	0.6585	+0.4	183.21	+1.6
59	8	32	16	1	3	0.5885	-10.3	190.17	+5.4
58	10	32	16	1	3	0.5169	-21.2	205.13	+13.7
60	12	32	16	1	3	0.4479	-31.7	227.14	+25.9
45	6	64	16	1	3	0.6785	+3.4	181.60	+0.7
43	8	64	16	1	3	0.6277	-4.3	182.40	+1.1
41	10	64	16	1	3	0.5762	-12.2	188.48	+4.5
55	12	64	16	1	3	0.5193	-20.9	200.89	+11.4
61	6	128	16	1	3	0.6829	+4.1	187.98	+4.2
40	8	128	16	1	3	0.6401	-2.4	186.93	+3.6
39	10	128	16	1	3	0.6029	-8.1	188.72	+4.6
38	12	128	16	1	3	0.5555	-15.3	197.09	+9.2

systems of varying mc and mb values. This flatness of the cost/performance surface causes both the rapid convergence and the error in the optimum prediction.

5.5.4.2 Experiment 6: Optimization on the $\ell m = 0$ hyperplane

Since, as Table 5.3 indicates, ERROR has no looping branches with a target distance of less than 128 bytes, loop-mode makes no contribution to system performance in the ranges of ib considered. Thus experiment 6, which runs ERROR on systems on the $\ell m = 0$ hyperplane, is essentially repeating experiment 5 with a different cost function, i.e., with the cost of loop-mode not included in the CPU cost. However, the procedure was started at (6, 16, 8, 1, 3) and gave rise to the convergence sequence listed in Table 5.21. The oscillation hypercube is still far away from the optimum along the mc dimension, for exactly the same reason as in experiment 5. Table 5.22 lists the sensitivity report for this experiment.

5.5.4.3 Analysis of the optimum architecture for ERROR

The optimum architecture for ERROR is thus seen to be (7, 64, 16, 1, 3) on the $\ell m = 0$ hyperplane. While there is quite a bit of interaction between the mb and mc parameters on the cost/performance surface, the surface is fairly orthogonal in the other parameters.

5.5.4.3.1 Instruction unit issues

As seen earlier, loop-mode contributes nothing to performance. Furthermore, the low percentage of branches ($\sim 2\%$) causes a high degree of prefetch (ib = 16) to be quite cost-effective, with superfluity becoming dominant only for higher values of ib (above 32).

Table 5.21 - Convergence Sequence for Experiment 6 on ERROR

Iteration	Reference System	Reference System Parameters				
		mc	mb	ib	fx	fl
0	2	6	16	8	1	3
1	12	8	32	16	1	2
2	13	7	16	8	1	3
3	14	6	32	16	1	3
4	15	8	16	8	1	3
5	16	7	32	16	1	3
6	17	9	64	8	1	3
7	18	10	32	16	1	3
8	19	11	64	8	1	3
9	20	12	128	16	1	3
10	21	13	64	32	1	3
11	22	12	128	64	1	3
12	23	11	64	32	1	3
13	18	10	32	16	1	3
14	19	11	64	8	1	3
15	24	10	128	16	1	3
16	19	11	64	8	1	3
17	24	10	128	16	1	3
Oscillation Hypercube		10:11	64:128	8:16	1	3

Table 5.22 - Sensitivity Report for Experiment 6 on ERROR

Optimum System : (7,64,16,1,3)

System	System Parameters					Performance		Cost		Cost/Performance	
	mc	mb	ib	fx	fl	Actual	%change from optimum	Actual	%change from optimum	Actual	%change from optimum
30	6	64	16	1	3	0.6785	+3.4	120.37	+4.2	177.40	+0.8
29	7	64	16	1	3	0.6561	0.0	115.52	0.0	176.08	0.0
28	8	64	16	1	3	0.6277	-4.3	111.64	-3.4	177.86	+1.0
16	7	32	16	1	3	0.6245	-4.8	112.94	-2.2	180.85	+2.7
29	7	64	16	1	3	0.6561	0.0	115.52	0.0	176.08	0.0
32	7	128	16	1	3	0.6692	+2.0	120.68	+4.5	180.33	+2.4
33	7	64	8	1	3	0.6333	-3.5	114.95	-0.5	181.50	+3.1
29	7	64	16	1	3	0.6561	0.0	115.52	0.0	176.08	0.0
34	7	64	32	1	3	0.6208	-5.4	116.66	+1.0	187.91	+6.7
29	7	64	16	1	3	0.6561	0.0	115.52	0.0	176.08	0.0
35	7	64	16	2	3	0.6561	0.0	117.23	+1.5	178.68	+1.5
36	7	64	16	1	2	0.2383	-63.7	112.67	-2.5	472.87	+168.6
29	7	64	16	1	3	0.6561	0.0	115.52	0.0	176.08	0.0

5.5.4.3.2 Execution unit issues

The proportion of fixed to floating point instructions, 0.045:1, is abnormally low in ERROR. Thus it comes as no surprise that fx and fℓ stay steadily at 1 and 3 respectively in the convergence sequence. Even more dramatic confirmation of this is obtained from the sensitivity report in Table 5.22. Increasing fx from 1 to 2 yields absolutely no performance increase, while reducing fℓ from 3 to 2, causes performance and cost/performance to degrade by phenomenal figures of 63.7% and 168.6% respectively.

5.5.4.3.3 Memory issues

For ERROR, both mc and mb seem to contribute roughly equally to performance. This is seen from the sensitivity report in which movements of 1 grid metric along either the mc or mb dimensions cause performance changes of the same order of magnitude - 2 to 5%. Since, however, their cost functions are different, mc and mb can be traded off against each other. Thus Table 5.19 shows that the (mc, mb) combination of (8, 64) is more cost-effective than (6, 32), as is (12, 64) over (10, 32). This wide range of choices for the pair of memory parameters to yield systems that have the same cost-effectiveness is what caused the optimization procedure to fail, as discussed earlier.

5.6 Efficiency of the Optimization Procedure

In this section, we estimate the efficiency of the procedure, using the definitions of Sec. 4.4.9. Table 5.23 illustrates the calculation of the η and η_L for the procedure in experiment 1a. Table 5.24 lists the η and η_L values for the experiments conducted. As expected, the efficiency is low in those experiments e.g. 2 and 5 where the oscillation hypercube was far from the optimum system since more sensitivity analysis is needed to identify the optimum. On the average, for the experiments

Table 5.23 - Efficiency Estimates for Experiment 1a on EIGEN

Number of low-level model calls
by the ideal procedure
= 3 x 3 x 3 x 2 x 2 = 108

Number of low-level model calls
by the optimization procedure = 302

Number of low-level model calls
by the grid evaluation procedure
= 14 x 6 x 3 x 3 x 3 = 2268

Efficiency of the optimization procedure:

$$\eta = \frac{108}{302} = 0.357$$

Efficiency of the grid evaluation procedure:

$$\eta_L = \frac{108}{2268} = 0.048$$

Table 5.24 - Efficiency Estimates for the Optimization Procedure

Experiment	Efficiency of optimization procedure (η)	Efficiency of grid evaluation procedure (η_L)
1a	0.357	0.048
1b	0.352	0.029
2	0.240	0.032
3	0.374	0.031
4	0.271	0.038
5	0.200	0.031
6	0.346	0.037

conducted, the ideal procedure is only 3.3 times more efficient than the optimization procedure which is 8.7 times more efficient than the grid evaluation procedure. Thus the optimization procedure is seen to be quite efficient in this case-study.

5.7 Some Architectural Conclusions

The analysis of the experiments conducted can be used to draw some broad conclusions with respect to the architecture.

1) The performance of the system is heavily dependent on the proportion of branches in the programs. Thus the performance of the optimum systems for the three traces is distinctly correlated with the percentage of branches in each, as shown in Table 5.25.

2) The best choice of instruction buffer size involves a tradeoff between increased instruction supply bandwidth due to instruction prefetch on the one hand, and superfluity and loop-mode on the other. The greater the proportion of branches, the smaller the prefetch needed. Loop-mode is cost-effective only with a high percentage of small program loops, where the branch decision and target fetching time form a small percentage of the loop execution time. At large buffer sizes, superfluity of prefetched instructions dominates.

3) The relative importance of the floating and fixed point execution units is in roughly the same proportion as the percentages of the two types of instructions in the programs.

4) Memory cycle time has a greater effect on performance than the number of memory banks.

Table 5.25 - Comparison of Various Systems

Program	% of branches	System	System Parameters					Performance		Cost/Performance	
			mc	mb	ib	fx	fl	Actual	%change from optimum	Actual	%change from optimum
		Normal	12	16	8	1	3	0.2567	-39.9	389.10	+33.4
EIGEN	8.20	Optimum	6	32	8	3	3	0.4272	0.0	291.72	0.0
		Design	5	32	16	3	3	0.4486	+5.0	291.83	+0.04
		Normal	12	16	8	1	3	0.1272	-55.7	784.93	+62.2
GAUSS	15.35	Optimum	4	16	16	3	3	0.2868	0.0	484.05	0.0
		Design	5	32	16	3	3	0.2753	-4.0	475.61	-1.7
		Normal	12	16	8	1	3	0.3106	-52.7	321.58	+78.2
ERROR	2.02	Optimum	7	64	16	1	3	0.6561	0.0	180.42	0.0
		Design	5	32	16	3	3	0.6832	+4.1	191.63	+6.2

5.7.1 A Final Design for the System

Let us assume that the program environment for which the system is designed is characterized by the three program traces used in the experiments. The final system design must then be a compromise between the three optimum systems arrived at for the three programs, with each optimum being weighted by the occurrence of the corresponding program in the environment. To illustrate this, we develop a compromise design system and evaluate its performance and cost/performance in the environment.

The flatness of the cost/performance surface for ERROR along the memory parameter dimensions, suggests that the memory design can be influenced largely by EIGEN and GAUSS. The compromise chosen between (mc = 6, mb = 32) for EIGEN and (4, 16) for GAUSS was (5, 32). The fℓ parameter was assigned the value 3, since all three programs require this. The dependence of EIGEN and GAUSS on fx, cause that to be assigned the value 3. In view of the ib tradeoff discussed in detailed in earlier sections, the compromise adopted was to fix ib at 16, with the degree of prefetch reduced to 8. Notice that this architecture is not in the space of systems considered by the optimization, where the degree of prefetch was always equal to ib. The dependence of EIGEN and GAUSS on ℓm, argue for ℓm = 1 in the design. Thus the final "design" system was (5, 32, 16, 3, 3) on the ℓm = 1 hyperplane.

The performance and cost/performance of the design system on the three programs is shown in Table 5.25, with the normal system and the respective optima also shown for comparison. The design system matches the optimum system for EIGEN in cost/performance and outperforms it, mainly due to the reduction in mc. Its cost/performance value for GAUSS is even better than that of the optimum system for GAUSS. This is possible because the

design system is not in the system space examined by the optimization procedure. The degradation in cost/performance on ERROR is due to the reduction in the degree of prefetch and the non-usage of the expensive fixed point unit.

CHAPTER 6

CONCLUSION

6.1 Summary of the Research

In this research, we have introduced the concept of a hierarchy of system performance models and discussed the characteristics and the construction of such a hierarchy. It was argued that such a hierarchy is a very useful tool for the cost-effective design of computer systems. A design procedure that uses this hierarchy was developed. The practicality and the usefulness of this procedure were demonstrated by applying it to the optimization of a complex computer system - the CPU-memory subsystem of the IBM System 360/91. In almost all the experiments, the optimization procedure converged, if not to the exact optimum system, at least to within a very near region of the optimum. The efficiency of the procedure is considerably more than that of the worst-case approach to system design, and is not substantially worse than that of the ideal procedure. Using the procedure yielded a great deal of insight into the behavior of the system.

6.2 Accomplishments of the Research

We summarize the main contribution of the research in this section.

1) Previous studies in the performance evaluation of computer systems have tended towards one of two extremes. At the one end are models in which reality has been sacrified for the sake of simplicity and mathematical tractability. While such studies do provide some insight into the system being modelled, their range and usefulness are severely limited because they are so far removed from realistic computer systems. At the other end are models which, because of their adherence to detail in the modelling of a specific system, have very little generality of use.

Our approach combines the tractability of the first kind with the accuracy of the other. It increases the range of applicability of the state-of-the-art performance modelling tools, by combining these synergistically into a powerful tool - the hierarchy of performance modelling tools. The main ingredients of this approach are the trilogy of calibration, validation and prediction, the proper use of which ensures accuracy as well as tractability. This is to be contrasted with most previous approaches to modelling, which have not laid enough emphasis on the iterative process of validation and recalibration before using the model for prediction.

2) The second major contribution is the embedding of a hierarchy of performance modelling tools into a system design (or optimization) procedure. The conflicting demands of standard iterative optimization procedures, viz, accuracy and ease of computation, are well matched by the attributes of the hierarchy. The practical problems with developing such an optimization procedure have been confronted and a number of issues brought to light. The success of the implemented procedure on the optimization of the case-study system is encouraging, and establishes the hierarchy as a viable design tool.

3) Some insight has been gained into the behaviour of highly pipelined single instruction stream CPU memory systems. Since the case-study system is an example of a highly complex computer system, the study leads us to believe that our understanding of complex systems can be improved by studies of this kind.

6.3 Suggestions for Further Research

We believe that our study opens up a vast area for further exploration in the performance evaluation field. We discuss some extensions

in the following subsections. First we discuss some specific improvements that can be made to the optimization procedure developed in Chapter 4. Then, we discuss more general ideas dealing with the extension and the application of the hierarchy concept.

6.3.1 Shortcomings of the Optimization Procedure and Suggested Remedies

We believe that the optimization procedure needs to be tuned further, to weed out some of the errors that come to light during the experiments. We now discuss some of the shortcomings of the procedure, and suggest some remedies.

6.3.1.1 Regression error

The experiments clearly show that in regions where the cost/performance surface has a small gradient along one dimension, the regression error must be less than this gradient value, for the procedure to converge reliably to the true optimum along that dimension. One possible way to reduce the regression error, i.e., obtaining a better fit to the performance surface, is to use a higher order regression model. Thus a quadratic model would express the response Y in terms of the factors $(X_1,\ldots X_m)$, using the functional form:

$$Y = \beta_0 + \sum_{i=1}^{m} \beta_i X_i + \sum_{i=1}^{m} \beta_{ii} X_i^2 + \sum_{i=1}^{m} \sum_{j=i+1}^{m} \beta_{ij} \cdot X_i X_j .$$

Statistical significance tests can be used to include only those factors and factor pairs that significantly affect the response. The cost of fitting and using such a model would still be an insignificant fraction of the low-level model cost.

We do not, however, recommend increasing the order of the analytical model indefinitely. Thus a 3rd order model could conceivably be worse than a 2nd order model, because it may introduce an oscillatory model surface, which creates a number of fictitious local optima. However, most performance curves do have a second order flavor, which argues for using a 2nd order model.

6.3.1.2 Choosing recalibration sets

In the procedure as implemented, when the current reference system has an extreme value along one dimension, e.g., $f\ell = 1$, only one nearest neighbor along that dimension, i.e. one with $f\ell = 2$, is needed to estimate the regression model coefficient along that dimension. Use of only one other point would disregard any non-linearities that occur on the cost/performance surface along that dimension. For example, Table 5.12, the sensitivity report for experiment 2, shows that near the optimum system both $f\ell = 1$ and $f\ell = 3$ are more cost-effective than $f\ell = 2$. Thus, if the reference system has $f\ell = 1$, it may never look beyond $f\ell = 2$, though $f\ell = 3$ may well be the optimum value. This could be remedied by using a higher order model, to better model the non-linearity of the surface. Thus a quadratic model would need at least three points along each dimension to compute the best fit, and $f\ell = 1,2,3$ would have to be considered.

6.3.1.3 Inability to maintain local perspective

When the reference system has moved into a new region, the new model should not be affected very much by old regions, i.e., local perspective should be maintained. In the procedure as implemented, this is not always possible. For example, in experiment 3 on GAUSS (see Table 5.14), by iteration 6 the reference system has moved to quite a different region from the initial region. However, to obtain β_4 the coefficient along the fx

dimension - a calibration system with fx = 2 is required, and the only one available in the calibration set is (12, 16, 8, 2, 3). However, including this system in the recalibration set, causes a member of other systems in the initial set to be included as well, since they are at the same distance from the new reference system. This distorts the local perspective enormously, especially along the most sensitive dimension - in this case, ib.

One possible remedy for this effect, is to generate extra calibration systems when in a new region. Typically, this can be done when it is observed that too many systems at too great a distance are being included in the recalibration set. This method can then be viewed as mingling the sensitivity and the optimization procedures.

6.3.1.4 Rigid movement rule

Requiring a change in every parameter of the reference system per iteration, is too rigid a movement rule. While, in the initial stages, it forces the procedure to roughly explore large regions of the system parameter space, it tends to cause unnecessary thrashing in the later stages, thus prolonging the convergence. For example, one of the iterations in experiment 1a on EIGEN, forced a movement along the mb dimension from 64 to 128, because the optimum predicted by the local optimization procedure was 64.33. It is clear that this difference from 64 could have been well within the regression error bounds of the procedure.

One possible remedy for this problem is to set a lower bound on the change for each parameter of the reference system. If the change predicted by the optimization procedure is less than the bound, the reference system would not be changed along that dimension. These bounds could be adaptively increased as the procedure converges.

6.3.1.5 Rigid stopping rule

Requiring the procedure to oscillate between two reference systems appears to be too rigid a stopping rule. In conjunction with the rigid movement rule, this caused quite a bit of thrashing in the early experiments on EIGEN.

A possible remedy is to monitor the change in cost/performance caused by the reference system movement. When this change goes below a limit, the procedure can be stopped.

6.3.1.6 Adaptive grid metrics

We believe that the adaptive approach to varying the mc grid metric is a reasonable one. It is in keeping with the philosophy of maintaining a global perspective in the initial stages and gradually narrowing the perspective as the procedure converges to the best region. However, the problem that arose in experiment 1c on EIGEN, may have to do with the actual implementation of the adaptive approach, as described in Sec. 4.5.2.

A technique similar to the one suggested in the last subsection may be used to control grid metric reduction. This would use percentage changes in cost/performance between successive reference systems to estimate the rate of convergence of the procedure. The grid metric change is computed as a function of this rate; in fact, it could actually be increased for small rates of convergence.

6.3.2 Further Research into the Hierarchy Concept and General Issues

The concept of a hierarchy of models for performance evaluation can be extended in a number of ways:

1) Hierarchies of more than two levels should be examined. Thus, in the two-level hierarchy considered in this study, an intermediate level that is less expensive than the low level model and has a larger range of validity than the high level model, could increase the cost-effectiveness of the procedure even further. In fact, in our research, the phase which consumed the most time and resources was the simulation runs of the low-level model. Introducing a level of intermediate complexity, e.g., a queueing model, would reduce the demands made on the low-level model even further. Consequently more, and larger, program traces could be experimented upon, to build a body of theory for architectures of this type. In general, the introduction of additional levels should be considered, if large regions of either complexity or cost are not covered by any model currently in the hierarchy.

2) In connection with (1) above, the hierarchy concept could be used on subsystems derived by a structural decomposition of low-level models. For example, building separate hierarchies for the memory and CPU would enable combinations of models of different levels of complexity for the different subsystems to be used as intermediate levels. Thus the intermediate level model could either be a CPU simulation model with an analytical memory model embedded in it, or vice versa, depending on the region being explored.

3) While quite a few computer system models have been proposed and analyzed, very little work has been done in modelling work loads to drive these system models. The proposal in Sec. 3.4.3.2, to generate synthetic control streams, based on statistical summaries of program environments

is a step in this direction, and should be further explored. Parameterizing workloads will enable studies of the system workload space analogous to this study of the system architecture space. Thus the optimum workload for a given system architecture and the sensitivity of the optimum system design to the workload parameters can be examined.

4) The techniques developed in this research can be applied to model and design computers at a higher level. Thus systems can be studied at the component level of processors, memories, I/O units, etc, besides the CPU function unit level studied in this thesis. It is our belief that the basic techniques could be applied, regardless of the level at which the system is studied.

5) Cost models other than the one used in this research should be investigated for their impact on system design. For example, using cache memories makes possible very low effective memory cycle times at low to moderate costs. The cost model used in this research did not allow for introducing a cache into the system. Parameterizing cost models, i.e., making the cost coefficients for the system parameters variable, will enable studies of the sensitivity of the optimum system design to changes in the cost coefficients. In an era of rapidly changing technology, such studies are of great importance to the system designer.

APPENDIX A

Theorem A.1

If $|A-C| < |A-B|$ then

a) $A > B$ iff $C > B$ and

b) $A < B$ iff $C < B$

Proof:

a) To prove that $A > B$ iff $C > B$:

a.1) Let $A > B$

Then $|A-C| < A-B$.

a.1.1) Let $A > C$.

Then $A-C < A-B$.

Therefore $-C < -B$ and thus $C > B$.

a.1.2) Let $A \leq C$.

Then $C \geq A > B$ and thus $C > B$.

Therefore if $A > B$ then $C > B$.

a.2) Let $C > B$.

Assume that $A \leq B$.

Then $|A-C| < B-A$.

a.2.1) Let $A > C$

Then $A > C > B$ and thus $A > B$, which is a contradiction.

a.2.2) Let $A \leq C$

Then $C-A < B-A$

Thus $C < B$ which is a contradiction.

Therefore, by contradiction if $C > B$ then $A > B$.

Therefore $A > B$ iff $C > B$.

b) To prove that $A < B$ iff $C < B$

b.1) Let $A < B$

Then $|A-C| < B-A$.

b.1.1) Let $A > C$.

Then $C < A < B$ and thus $C < B$,

b.1.2) Let $A \leq C$.

Then $C-A < B-A$ and

thus $C < B$.

Therefore if $A < B$ then $C < B$.

b.2) Let $C < B$.

Assume that $A \geq B$

Then $|A-C| < A-B$.

b.2.1) Let $A > C$.

Then $A-C < A-B$.

Therefore $-C < -B$ and thus $C > B$, which is a contradiction.

b.2.2) Let $A \leq C$.

Then $C \geq A \geq B$ and thus $C \geq B$, which is a contradiction.

Therefore, by contradiction, if $C < B$ then $A < B$.

Therefore $A < B$ iff $C < B$.

Q.E.D

APPENDIX B

B.1 Introduction

In this appendix, we present a listing of the simulator of the control stream model. The simulator was written in the SIMULA language, an excellent introduction to which is given in [BIR73]. The system used was the SIMULA-10 system [BIR74], developed at the Swedish National Defense Research Institute in Stockholm. The system was run on the DEC-10 at the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. Execution times of simulator runs ranged from 12 to 40 minutes.

In the listing that follows, comment statements begin with a '!' and end with the first ';' thereafter.

```
B1      00050   BEGIN !TDMOD;
        00100   !THE CONTROL STREAM MODEL SIMULATOR.
        00150   IT OUTPUTS THE INSTRUCTION THROUGHPUT AND OTHER USEFUL STATISTICS INTO THE SUMMARY FILE.
        00200   IT HAS 6 PARAMETERS:
        00250           CYCLE : THE MEMORY CYCLE TIME.
        00300           BANKS : THE NUMBER OF MEMORY BANKS.
        00350           BUFFERS  : THE SIZE OF THE INSTRUCTION BUFFER.
        00400           FIXARCH : THE FIXED POINT UNIT ARCHITECTURE.
        00450           FLARCH : THE FLOATING POINT UNIT ARCHITECTURE.
        00500           BRARCH: =TRUE IF LOOPMODE IS TO BE INCLUDED IN THE SYSTEM.;
        00550
        00600       TEXT INBUF,OUTBUF1;
        00650       INTEGER I,J,K,L,PC,BUFFERS,BANKS,CYCLE,ACCESS,FIXARCH,FLARCH;
        00700       BOOLEAN BRARCH;
        00750       REF(INFILE) INPUT; REF(PRINTFILE) OUTPUT;
        00800
        00850       !GET THE MODEL PARAMETER VALUES;
        00900       OUTTEXT("MEMORY CYCLE TIME:"); BREAKOUTIMAGE; CYCLE:=ININT;
        00950       ACCESS:=CYCLE*5/6;
        01000       OUTTEXT("NUMBER OF MEMORY BANKS:"); BREAKOUTIMAGE; BANKS:=ININT;
        01050       OUTTEXT("INSTRUCTION BUFFER SIZE:"); BREAKOUTIMAGE; BUFFERS:=ININT;
        01100       OUTTEXT("FIXED POINT UNIT ARCHITECTURE:"); BREAKOUTIMAGE; FIXARCH:=ININT;
        01150       OUTTEXT("FLOATING POINT UNIT ARCHITECTURE:"); BREAKOUTIMAGE; FLARCH:=ININT;
        01200       OUTTEXT("LOOP MODE:"); BREAKOUTIMAGE; I:=ININT;
        01250       BRARCH:=IF I=0 THEN FALSE ELSE TRUE;
        01300
        01350       !OPEN THE INPUT AND OUTPUT FILES;
        01400       INBUF:-BLANKS(50); INPUT:-NEW INFILE("INPUT *"); INPUT.OPEN(INBUF);
        01450       OUTBUF1:-BLANKS(132); OUTPUT:-NEW PRINTFILE("SUMARY *"); OUTPUT.OPEN(OUTBUF1);
        01500
B2      01550       SIMULATION BEGIN
        01600           INTEGER BUFCOUNTER,PROCCOUNT,LOOPGEN,NORMGEN,MEMCOUNT,NORMTERM,DUMTERM,ABTERM,LASTSST;
        01650           INTEGER ABTIME,LASTGEN,LASTTERM,LASTINST,LOOPSIZE,ACTIVETAG,NEXTTAG,GLICH,GLSTART,GLTIME;
        01700           INTEGER SERIAL,PIPELINE,DATAFLOW,INDEP;
        01750           INTEGER IGEN,ITERM,NTERM;
        01800           INTEGER TYPE,OPCODE,STORE,CCTAG,BRADD,SNKMAS,SRCMAS,SNKSLV,SRCSLV,INDREG,MEMADD,BRDIST,TAKEBR;
        01850           INTEGER RR,RX,SI,BRANCH,TRACEND,FLOAT,SVC,BALR,BC15,BCR15,BCT,FETSTR,FETCH,SISTR;
        01900           INTEGER FXLA,FXST,FXLD,FXSPLD,FXAD,FXCMP,FXMUL,FXDIV,FLLD,FLSPLD,FLCMP,FLMUL,FLST,FLDIV;
        01950           INTEGER ARRAY EXEC[0:56],TIMEHIST[1:3,0:1],SSTSLAVE[0:451];
        02000           BOOLEAN CONDCODE,CONDMODE,NOEXEC,LOOPMODE,ABORT,FXIUFREE,FXEUFREE,FLIUFREE,FLADFREE,FLMDFREE,ENDPROG;
        02050           BOOLEAN ARRAY BANKFREE[1:BANKS],SST[1:451];
        02100           REF(HEAD) IBUFCH,FXIUCH,FXEUCH,FLIUCH,FLADCH,FLMDCH,BCONDCH;
        02150           REF(HEAD) ARRAY SSTCH[1:451],BANKCH[1:BANKS];
        02200           REF(INSTRUCTION) SSTWAITER;
        02250
        02300           PROCEDURE TIMESTATS(WHICHSTAT,BAR);
        02350           !THIS PROCEDURE MAINTAINS THRUPUT STATISTICS;
        02400           INTEGER WHICHSTAT,BAR;
B3      02450           BEGIN
        02500               !UPDATE THE NO.-OF-PROCESSES COLUMN;
        02550               TIMEHIST[WHICHSTAT,0]:=TIMEHIST[WHICHSTAT,0]+1;
        02600               !UPDATE THE TIME INTEGRAL COLUMN;
```

```
                    TIMEHIST[WHICHSTAT,1]:=TIMEHIST[WHICHSTAT,1]+BAR;
E3                  END OF TIMESTATS;

                    PROCESS CLASS CONTROL(PRIORITY);
                    !THIS CLASS IS THE ROOT OF THE PROCESS TREE THAT DEFINES THE PROCESSES
                    THAT FORMS THE CORE OF THE SIMULATION;
                    REAL PRIORITY; !PRIORITY=TIME OF CREATION.;
B4                  BEGIN

                        PROCEDURE SCHEDULE(TRANS,DISPLACEMENT);
                        !THIS PROCEDURE INSERTS THE PROCESS TRANS INTO THE CORRECT PLACE
                        IN THE EVENT CHAIN,DEPENDING ON ITS PRIORITY AND THE TIME OF
                        NEXT ACTIVATION (= CURRENT TIME + DISPLACEMENT);
                        REF(CONTROL) TRANS; REAL DISPLACEMENT;
B5                      BEGIN
                            REF(CONTROL) ONE,TWO; REAL ABSTIME;
                            ABSTIME:=TIME+DISPLACEMENT; !THE TIME OF NEXT ACTIVATION;
                            ONE:-CURRENT.NEXTEV; TWO:-CURRENT; !2 POINTERS TO CHASE THE EVENT LIST;
                            WHILE (IF ONE=/=NONE THEN ONE.EVTIME<ABSTIME OR (ONE.EVTIME=ABSTIME AND ONE.PRIORITY<=TRANS.PRIORITY
) ELSE FALSE) DO
                            !IF EVENTS ARE SHCEDULED TO OCCUR BEFORE THIS OR PRIORITY OF THIS ONE IS LESS, CARRY ON;
B6                          BEGIN
                                TWO:-ONE; ONE:-ONE.NEXTEV;
E6                          END;
                            !HAVE FOUND WHERE THIS PROCESS SHOULD GO IN THE EVENT CHAIN;
                            !IF SOME PROCESS IS TO OCCUR AT THE SAME TIME, INSERT THIS AFTER THAT ONE;
                            IF TWO.EVTIME=ABSTIME THEN REACTIVATE TRANS AFTER TWO
                            !IF NOT,INSERT IT AHEAD OF PROCESSES SCHEDULED FOR LATER ON;
                            ELSE REACTIVATE TRANS AT ABSTIME PRIOR;
E5                      END OF SCHEDULE;

                        PROCEDURE LOUNGE(CHAIN);
                        !THIS PROCEDURE PUTS THE PROCESS INTO WAITING STATE ON A CHAIN;
                        REF(HEAD) CHAIN;
                        IF CHAIN==NONE THEN PASSIVATE ELSE
B7                      BEGIN
                            WAIT(CHAIN); CURRENT.OUT;
E7                      END OF LOUNGE;

                        PROCEDURE NEXTGUY(CHAIN,FLAG);
                        NAME FLAG; BOOLEAN FLAG; REF(HEAD) CHAIN;
                        !THIS PROCEDURE SCHEDULES THE HEAD OF THE WAITING LIST "CHAIN" FOR PROCESSING BY A RESOURCE.
                        IT THEN SETS THE RESOURCE BUSY FLAG.;
B8                      BEGIN
                            REF(CONTROL) TRANS;
                            TRANS:-CHAIN.FIRST;
                            IF TRANS=/=NONE THEN
B9                          BEGIN
                                SCHEDULE(TRANS,0); FLAG:=FALSE;
E9                          END ELSE FLAG:=TRUE;
E8                      END OF NEXTGUY;

                        PROCCOUNT:=PROCCOUNT+1; !UP THE TOTAL PROCESS COUNT;
```

```
        05250               INNER; !PERFORM THE PROCESS FUNCTIONS;
        05300               PROCCOUNT:=PROCCOUNT-1; !DOWN THE ACTIVE PROCESS COUNT;
        05350               IF PROCCOUNT=0 THEN ACTIVATE MAIN; !IF ALL PROCESSES ARE DONE,TERMINATE THE SIMULATION;
E4      05400           END OF CONTROL;
        05450
        05500           CONTROL CLASS CPUCONTROL;
B10     05550           BEGIN
        05600           !THE NEXT LEVEL OF THE TREE. WILL SPROUT BOTH INSTRUCTION AND MEMREF PROCESSES;
        05650               INTEGER ARRAY PARM[1:12]; !HOLDS THE INSTRUCTION DESCRIPTORS;
        05700
        05750               PROCEDURE RELEASE(CHAIN,FLAG);
        05800               !THIS PROCEDURE RELEASES PROCESSES QUEUED ON CHAIN AND SETS FLAG;
        05850               NAME FLAG; BOOLEAN FLAG; REF(HEAD) CHAIN;
B11     05900               BEGIN
        05950                   REF(CPUCONTROL) TRANS; INTEGER I;
        06000                   FLAG:=TRUE; !SET THE FLAG;
        06050                   TRANS:-CHAIN.FIRST;
        06100                   WHILE TRANS=/=NONE DO
        06150                   !TRAVERSE THE CHAIN, RELEASING PROCESSES;
B12     06200                   BEGIN
        06250                       SCHEDULE(TRANS,0); TRANS:-TRANS.SUC;
E12     06300                   END;
E11     06350               END OF RELEASE;
        06400
        06450               PROCEDURE FREESLOT(POINTER);
        06500               !THIS PROCEDURE FREES A SLOT IN THE SYSTEM STATUS TABLE TO BE USED BY A LATER INSTRUCTION;
        06550               INTEGER POINTER;
        06600               IF PARM[POINTER]\=INDEP THEN
        06650               !NEED TO FREE THE SLOT ONLY IF IT IS NOT THE PERENNIALLY FREE SLOT;
B13     06700               BEGIN
        06750                   SSTSLAVE[PARM[POINTER]]:=SSTSLAVE[PARM[POINTER]]-1;
        06800                   !IF NO MORE SLAVES ARE WAITING ON THIS SLOT,FREE ANY NEW INSTRUCTIONS WAITING TO GRAB THIS SLOT.;
        06850                   IF SSTSLAVE[PARM[POINTER]]=0 AND SSTWAITER=/=NONE AND LASTSST=PARM[POINTER] THEN
B14     06900                   BEGIN
        06950                       SCHEDULE(SSTWAITER,0); SSTWAITER:-NONE;
E14     07000                   END;
E13     07050               END OF FREESLOT;
        07100
        07150               !INTER-GENERATION STATISTICS FOR INSTRUCTION AND OPERAND PROCESSES.;
        07200               TIMESTATS(IGEN,TIME-LASTGEN); LASTGEN:=TIME;
        07250               INNER;
E10     07300           END OF CPUCONTROL;
        07350
        07400           CPUCONTROL CLASS INSTRUCTION(BUFPOS);
        07450           INTEGER BUFPOS; !POSITION IN THE INSTRUCTION BUFFER WHERE THE INSTRUCTION WILL RESIDE.;
        07500           !THE INSTRUCTION PROCESS;
B15     07550           BEGIN
        07600               INTEGER I,SYNC; REF(OPERAND) OFFSPRING; BOOLEAN MEMOP;
        07650
        07700               PROCEDURE PREDECODE;
        07750               !THIS PROCEDURE DOES THE SCHEDULING PRIOR TO INSTRUCTION DECODING;
B16     07800               BEGIN
```

```
                    REF(INSTRUCTION) TRANS;
                    !SCHEDULE NEXT INSTRUCTION FETCH ONLY IF THE END OF THE TRACE HAS NOT BEEN REACHED.;
                    IF \ENDPROG THEN SCHEDULE(NEW INSTRUCTION(TIME,BUFCOUNTER),0);
                    !BUFCOUNTER POINTS AT NEXT INSTRUCTION IN THE BUFFER TO BE SCHEDULED FOR EXECUTION.;
                    BUFCOUNTER:=1+MOD(BUFCOUNTER,BUFFERS); TRANS:-IBUFCH.FIRST;
                    !SCHEDULE SUCCESSOR INSTRUCTION FOR EXTRACTION FROM IBUF;
                    WHILE(IF TRANS==NONE THEN FALSE ELSE TRANS.BUFPOS\=BUFCOUNTER) DO TRANS:-TRANS.SUC;
                    IF TRANS=/=NONE THEN SCHEDULE(TRANS,0);
E16                 END OF PREDECODE;

                    !IF NOT IN LOOPMODE, START OFF WITH A MEMORY REFERENCE TO FETCH THE INSTRUCTION;
                    IF \LOOPMODE THEN
B17                 BEGIN
                        !NORMAL INSTRUCTION GENERATION STATISTICS;
                        NORMGEN:=NORMGEN+1;
                        !START OFF MEMORY FETCH AND BUMP PROGRAM COUNTER VALUE (IN BANKS).;
                        SCHEDULE(NEW MEMREF(TIME,PC,CURRENT),0); PC:=1+MOD(PC,BANKS);
                        !WAIT FOR IT TO RETURN FROM MEMORY;
                        LOUNGE(NONE);
E17                 END ELSE LOOPGEN:=LOOPGEN+1; !LOOPMODE INSTRUCTION GENERATION STATS.;
                    !WHEN INSTRUCTION FETCH IS OVER,WAIT UNTIL YOUR TURN TO BE DECODED.;
                    IF BUFPOS\=BUFCOUNTER THEN LOUNGE(IBUFCH);
                    SCHEDULE(CURRENT,1); !INSTRUCTION EXTRACTION FROM THE BUFFER;

                    !CHECK IF YOU ARE FOLLOWING A BRANCH THAT IS GOING TO SWITCH STREAMS;
                    IF ABORT AND PRIORITY<ABTIME THEN
B18                 BEGIN
                        ABTERM:=ABTERM+1; !ABORT STATISTICS;
                        PREDECODE; !SCHEDULE FOLLOWING INSTRUCTIONS;
                        !THIS INSTRUCTION HAS BEEN ABORTED.;
E18                 END ELSE
B19                 BEGIN
                        !THE FIRST INST STARTED AFTER THE DECISION OF THE BRANCH TURNS OFF THE ABORT SWITCH;
                        ABORT:=FALSE;
                        !IS IT A DUMMY INSTRUCTION CONDITIONALLY ISSUED FOLLOWING A CONDITIONAL BRANCH?;
                        IF CONDMODE AND NOEXEC THEN
B20                     BEGIN
                            DUMTERM:=DUMTERM+1; !DUMMY INSTRUCTION STATISTICS;
                            !TIME IN THE DECODER;
                            PREDECODE; SCHEDULE(CURRENT,1);
                            !WAIT IN THE BCOND CHAIN AND THEN DIE.;
                            IF CONDMODE THEN LOUNGE(BCONDCH);
E20                     END ELSE

                        IF ENDPROG THEN PREDECODE ELSE
                        !IF THE END OF THE PROGRAM HAS NOT YET BEEN REACHED AND IT IS A NORMAL INSTRUCTION;
B21                     BEGIN
                            !NOW READ THE INSTRUCTION DESCRIPTORS;
                            INSPECT INPUT DO
B22                         BEGIN
                                INIMAGE; INCHAR; INCHAR; !SKIP PAST 2 DUMMY CHARACTERS AT THE BEGINNING OF THE LINE.;
                                FOR I:=1,2,3,4 DO
```

```
        10450                       !THE RESOURCE USAGE PARAMETERS;
B23     10500                       BEGIN
        10550                           PARM[I]:=ININT; INCHAR;
E23     10600                       END;
        10650                       FOR I:=9,10,11,12 DO
        10700                       !OTHER PARAMETERS;
B24     10750                       BEGIN
        10800                           PARM[I]:=ININT; INCHAR;
E24     10850                       END;
        10900                       !CONVERT RAW MEMORY ADDRESSES INTO BANK NUMBERS;
        10950                       IF PARM[TYPE]=BRANCH THEN PARM[BRADD]:=1+MOD(PARM[BRADD]//8,BANKS)
        11000                       ELSE PARM[MEMADD]:=1+MOD(PARM[MEMADD]//8,BANKS); !THE BANK REFERENCED;
        11050                       IF NOT(PARM[TYPE]=TRACEND OR PARM[TYPE]=BRANCH OR PARM[OPCODE]=SVC) THEN
B25     11100                       BEGIN
        11150                       !DOES IT NEED DATA DEPENDENCY PARAMETERS;
        11200                           INIMAGE;
        11250                           FOR I:=5,6,7,8 DO
B26     11300                           BEGIN
        11350                               PARM[I]:=ININT; INCHAR;
E26     11400                           END;
        11450                           !A GLICH TO AVOID SST OVERFLOW ON THE SINK OPERAND SIDE.;
        11500                           !IF THIS SLOT HAS SOME DEPENDENT SLAVES STILL WAITING FOR IT.;
        11550                           IF \SST[PARM[SNKSLV]] OR SSTSLAVE[PARM[SNKSLV]]>0 THEN
        11600                           !OVERFLOW HAS OCCURRED. STOP TILL THIS SLOT IS FREED;
B27     11650                           BEGIN
        11700                               GLICH:=GLICH+1; GLSTART:=TIME; !GLICH STATISTICS;
        11750                               !IF THE SLOT IS NOT YET FREE,WAIT;
        11800                               IF \SST[PARM[SNKSLV]] THEN LOUNGE(SSTCH[PARM[SNKSLV]]);
        11850                               !IF THE SLOT IS FREE BUT ITS DEPENDENTS HAVE NOT YET SEEN THAT,WAIT;
        11900                               IF SSTSLAVE[PARM[SNKSLV]]>0 THEN
B28     11950                               BEGIN
        12000                                   !MARK THIS PROCESS;
        12050                                   SSTWAITER:-CURRENT; LASTSST:=PARM[SNKSLV]; LOUNGE(NONE);
E28     12100                               END;
        12150                               GLTIME:=GLTIME+TIME-GLSTART;
E27     12200                           END OF SNKSLV GLICH;
        12250                           !TO AVOID OVERFLOW FROM THE SOURCE OPERAND SIDE,DO EXACTLY THE SAME.;
        12300                           IF \SST[PARM[SRCSLV]] OR SSTSLAVE[PARM[SRCSLV]]>0 THEN
B29     12350                           BEGIN
        12400                               GLICH:=GLICH+1; GLSTART:=TIME;
        12450                               IF \SST[PARM[SRCSLV]] THEN LOUNGE(SSTCH[PARM[SRCSLV]]);
        12500                               IF SSTSLAVE[PARM[SRCSLV]]>0 THEN
B30     12550                               BEGIN
        12600                                   SSTWAITER:-CURRENT; LASTSST:=PARM[SRCSLV]; LOUNGE(NONE);
E30     12650                               END;
        12700                               GLTIME:=GLTIME+TIME-GLSTART;
E29     12750                           END OF SRCSLV GLICH;
        12800
        12850                           !NOW GRAB THE AVAILABLE SLOTS AND MARK THOSE OPERANDS AS UNAVAILABLE.;
        12900                           IF PARM[SNKSLV]\=INDEP THEN SST[PARM[SNKSLV]]:=FALSE;
        12950                           IF PARM[SNKMAS]\=INDEP THEN SSTSLAVE[PARM[SNKMAS]]:=SSTSLAVE[PARM[SNKMAS]]+1;
        13000                           IF PARM[SRCSLV]\=INDEP THEN SST[PARM[SRCSLV]]:=FALSE;
```

```
                                    IF PARM[SRCMAS]\=INDEP THEN SSTSLAVE[PARM[SRCMAS]]:=SSTSLAVE[PARM[SRCMAS]]+1;
E25                             END;
                                !ONE MORE SLAVE FOR THE INDEX REGISTER MASTER SLOT IN THE SST.;
                                IF PARM[INDREG]\=INDEP THEN SSTSLAVE[PARM[INDREG]]:=SSTSLAVE[PARM[INDREG]]+1;
E22                         END OF INPUT;

                            !HAS THE END OF THE TRACE BEEN REACHED?;
                            IF PARM[TYPE]=TRACEND THEN
B31                         BEGIN
                                !STOP ACTIVITIES BUT FLUSH THE BUFFER FIRST.;
                                ENDPROG:=TRUE; PREDECODE;
E31                         END ELSE
B32                         BEGIN
                                !IS THERE AN INDEX REGISTER DEPENDENCY?;
                                IF PARM[INDREG]\=INDEP THEN
B33                             BEGIN
                                    !RESOLVE INDEX REGISTER DEPENDENCY;
                                    IF \SST[PARM[INDREG]] THEN LOUNGE(SSTCH[PARM[INDREG]]);
                                    FREESLOT(INDREG);
E33                             END OF INDEX REGISTER DEPENDENCY;

                                IF PARM[OPCODE]=SVC OR (PARM[TYPE]=BRANCH AND (PARM[OPCODE]<=BALR
                                 OR PARM[OPCODE]=BC15 OR PARM[OPCODE]=BCR15)) THEN
                                !AN UNCONDITIONAL BRANCH;
B34                             BEGIN
                                    !IF THE SYSTEM IS IN CONDITIONAL MODE,DO NOT ISSUE THIS INSTRUCTION;
                                    IF CONDMODE THEN LOUNGE(BCONDCH);
                                    !SET THE STAGE FOR A MASS EXTERMINATION OF FOLLOWING INSTRUCTIONS;
                                    ABORT:=TRUE; ABTIME:=TIME;
                                    !SWITCH INST STREAM FETCHING AND NULLIFY LOOPMODE;
                                    PC:=PARM[BRADD]; LOOPMODE:=FALSE;
                                    !SCHEDULE FOLLOWING INSTRUCTIONS AND GET DECODED.;
                                    PREDECODE; SCHEDULE(CURRENT,1);
E34                             END ELSE

                                !IF IT IS A CONDITIONAL BRANCH;
                                IF PARM[TYPE]=BRANCH THEN
B35                             BEGIN
                                    !WAIT IF THE SYSTEM IS IN CONDITIONAL MODE;
                                    IF CONDMODE THEN LOUNGE(BCONDCH);
                                    !IS THIS BRANCH GOING TO SET LOOP MODE?;
                                    IF BRARCH AND \LOOPMODE AND PARM[TAKEBR]=1 AND PARM[BRDIST]<=LOOPSIZE THEN LOOPMODE:=TRUE;
                                    !IS IT AN INDEXING BRANCH INSTRUCTION?;
                                    IF PARM[OPCODE]>=BCT THEN
B36                                 BEGIN
                                        SCHEDULE(CURRENT,2); !TIME FOR DECODING AND BRANCH DECISION ARITHMETIC;
                                        PREDECODE; !SCHEDULE FOLLOWERS;
E36                                 END ELSE
                                    !IF IT IS DEPENDENT ON THE CONDITION CODE;
B37                                 BEGIN
                                        SCHEDULE(CURRENT,1); PREDECODE; !DECODING TIME;
                                        !IF THE CONDITION CODE HAS NOT BEEN SET,ISSUE IN  CTIONS IN CONDITIONAL MODE.;
```

```
        15650                                   IF \CONDCODE THEN
B38     15700                                   BEGIN
        15750                                       CONDMODE:=TRUE; !SET CONDITIONAL MODE OF ISSUING;
        15800                                       !IF THE INST STREAM IS GOING TO BE SWITCHED,ISSUE DUMMY INSTRUCTIONS;
        15850                                       NOEXEC:=(IF (PARM[TAKEBR]=1 AND LOOPMODE) OR
        15900                                        (PARM[TAKEBR]=0 AND \LOOPMODE) THEN FALSE ELSE TRUE);
        15950                                       LOUNGE(BCONDCH); !WAIT FOR CONDITION CODE TO BE SET;
        16000                                       CONDMODE:=NOEXEC:=FALSE;
        16050                                       !UPDATE SYSTEM STATUS (REGARDING CONDITION CODE SETTING INSTRUCTIONS)
        16075                                       CHANGED DURING CONDITIONAL MODE;
        16100                                       IF ACTIVETAG\=NEXTTAG THEN
B39     16150                                       BEGIN
        16200                                           CONDCODE:=FALSE; ACTIVETAG:=NEXTTAG;
E39     16250                                       END;
E38     16300                                   END;
E37     16350                               END OF CONDITION-CODE-DEPENDENT INSTRUCTIONS;
        16400                               !HAS THE INSTRUCTION STREAM TO BE SWITCHED?;
        16450                               IF (\LOOPMODE AND PARM[TAKEBR]=1) OR (LOOPMODE AND PARM[TAKEBR]=0) THEN
B40     16500                               BEGIN
        16550                                   ABORT:=TRUE; ABTIME:=TIME; !ABORT THE REST OF IBUF;
        16600                                   !IF LOOPMODE IS ON CALCULATE THE NEW STREAM ADDRESS FROM THE BRADD AND THE BRDIST PA
RAMETERS;
        16650                                   PC:=IF \LOOPMODE THEN PARM[BRADD] ELSE 1+MOD(PARM[BRADD]+PARM[BRDIST]//8,BANKS);
        16700                                   LOOPMODE:=FALSE; !TURN THE LOOPMODE SWITCH OFF;
E40     16750                               END;
E35     16800                           END ELSE
        16850
        16900                           !FOR NON-BRANCHING INSTRUCTIONS;
B41     16950                           BEGIN
        17000                               PREDECODE; SCHEDULE(CURRENT,1); !DECODING TIME;
        17050                               !CAN IT SET THE CONDITION CODE?;
        17100                               IF PARM[CCTAG]>0 THEN
B42     17150                               BEGIN
        17200                                   NEXTTAG:=PARM[CCTAG]:=1+MOD(NEXTTAG,2**10);
        17250                                   !IF NOT IN CONDITIONAL MODE, THIS INSTRUCTION NOW HAS THE VALID CONDITION-CODE SETTI
NG TAG;
        17300                                   IF \CONDMODE THEN
B43     17350                                   BEGIN
        17400                                       CONDCODE:=FALSE; ACTIVETAG:=NEXTTAG;
E43     17450                                   END;
E42     17500                               END;
        17550                               !DOES IT NEED AN OPERAND FROM MEMORY?;
        17600                               IF (PARM[SRCSLV]\=INDEP AND (PARM[TYPE]=RX AND PARM[STORE]\=1))
        17650                                OR (PARM[TYPE]=SI AND (PARM[OPCODE]=FETSTR OR PARM[OPCODE]=FETCH)) THEN
B44     17700                               BEGIN
        17750                                   !SCHEDULE AN OPERAND PROCESS TO FETCH THE OPERAND FROM MEMORY.;
        17800                                   MEMOP:=TRUE; OFFSPRING:-NEW OPERAND(TIME,CURRENT);
        17850                                   SCHEDULE(OFFSPRING,0);
E44     17900                               END;
        17950                               !WAIT IF BEEN ISSUED IN CONDITIONAL MODE;
        18000                               IF CONDMODE THEN LOUNGE(BCONDCH);
        18050
        18100                               !FIXED POINT INSTRUCTION;
        18150                               IF PARM[OPCODE]<FLOAT THEN
```

```
B45     18200                           BEGIN
        18250                               SCHEDULE(CURRENT,1); !TRANSFER TO FIXED POINT UNIT;
        18300                               !IF THE FIXED POINT DECODER IS FREE,GRAB IT ELSE WAIT.;
        18350                               IF NOT FXIUFREE THEN LOUNGE(FXIUCH) ELSE FXIUFREE:=FALSE;
        18400                               SCHEDULE(CURRENT,1); !FIXED POINT DECODING;
        18450                               !DOES IT NEED A REGISTER SOURCE OPERAND?;
        18500                               IF PARM[SRCSLV]\=INDEP AND (PARM[TYPE]=RR OR (PARM[TYPE]=RX AND PARM[STORE]=1)) THEN
B46     18550                               BEGIN
        18600                                   !SCHEDULE AN OPERAND PROCESS TO FETCH THE REGISTER OPERAND.;
        18650                                   OFFSPRING:-NEW OPERAND(TIME,CURRENT); SCHEDULE(OFFSPRING,0);
E46     18700                               END ELSE
        18750                               IF \MEMOP THEN
B47     18800                               BEGIN
        18850                                   !IF THIS IS A 1-OPERAND INSTRUCTION,HALF THE SYNCHRONIZATION IS OVER.;
        18900                                   SYNC:=1; FREESLOT(SRCMAS);
E47     18950                               END;
        19000                               !DOES IT NEED A SINK OPERAND AS WELL?;
        19050                               IF NOT(PARM[SNKSLV]=INDEP OR PARM[OPCODE]=FXLA OR PARM[OPCODE]=FXST
        19100                                OR (PARM[TYPE]<SI AND PARM[OPCODE]<=FXLD) OR PARM[TYPE]=SI) THEN
B48     19150                               BEGIN
        19200                                   !IS THE SINK OPERAND AVAILABLE?;
        19250                                   IF SST[PARM[SNKMAS]] THEN
B49     19300                                   BEGIN
        19350                                       FREESLOT(SNKMAS); !FREE THE SINK SLOT.;
        19400                                       SCHEDULE(CURRENT,1); !OPERAND FORWARDING TIME;
        19450                                       !IF DATA-FLOW ARCHITECTURE,RELEASE DECODER;
        19500                                       IF FIXARCH=DATAFLOW THEN NEXTGUY(FXIUCH,FXIUFREE);
E49     19550                                   END ELSE
        19600                                   !IF THE SINK OPERAND IS NOT AVAILABLE;
B50     19650                                   BEGIN
        19700                                       !FOR DATA-FLOW ARCHITECTURE,RELEASE DECODER;
        19750                                       IF FIXARCH=DATAFLOW THEN NEXTGUY(FXIUCH,FXIUFREE);
        19800                                       !WAIT FOR OPERAND TO BECOME AVAILABLE;
        19850                                       LOUNGE(SSTCH[PARM[SNKMAS]]); FREESLOT(SNKMAS);
        19900                                       !FOR NON-DATA-FLOW ARCHITECTURE,TRANSFER THE OPERAND AFTER IT IS AVAILABLE.;
        19950                                       IF FIXARCH\=DATAFLOW THEN SCHEDULE(CURRENT,1);
E50     20000                                   END;
        20050                                   !IF IT IS A COMPARE INSTRUCTION,THE OPERAND IS AVAILABLE IMMEDIATELY.;
        20100                                   IF PARM[OPCODE]=FXCMP THEN RELEASE(SSTCH[PARM[SNKSLV]],SST[PARM[SNKSLV]]);
E48     20150                               END ELSE
B51     20200                               BEGIN
        20250                                   !IF IT IS NOT A STORE INSTRUCTION THE SINK SLOT CAN BE FREED NOW ITSELF.;
        20300                                   IF \(PARM[OPCODE]=FXST OR (PARM[TYPE]=SI AND PARM[OPCODE]=SISTR)) THEN FREESLOT(
SNKMAS);
        20350
        20400                                   !IF DATA-FLOW ARCHITECTURE,LET THE DECODER GO;
        20450                                   IF FIXARCH=DATAFLOW THEN NEXTGUY(FXIUCH,FXIUFREE);
E51     20500                               END;
        20550
        20600                               !SYNCHRONIZE WITH THE OPERAND PROCESS;
        20650                               SYNC:=SYNC+1;
HERE FIRST.;                                !IF THE 2 PATHS HAVE NOT YET MERGED,WAIT ELSE RELEASE THE OPERAND PROCESS THAT GOT T
        20700
        20750                               IF SYNC<2 THEN LOUNGE(NONE) ELSE
                                            IF OFFSPRING=/=NONE THEN SCHEDULE(OFFSPRING,0);
```

```
        20800                           !IF A MEMORY OPERAND IS NEEDED IN A NON-DATA-FLOW SYSTEM,FORWARDING TIME;
        20850                           IF FIXARCH\=DATAFLOW AND MEMOP THEN SCHEDULE(CURRENT,1);
        20900                           IF PARM[TYPE]<SI AND (PARM[OPCODE]=FXLD OR PARM[OPCODE]=FXLA) THEN
        20950                           !IF IT IS A SIMPLE LOAD INTO A REGISTER,IT HAS ALLREADY BEEN DONE;
B52     21000                           BEGIN
        21050                               !FOR NON-DATA-FLOW SYSTEMS,RELEASE DECODER;
        21100                               IF FIXARCH\=DATAFLOW THEN NEXTGUY(FXIUCH,FXIUFREE);
E52     21150                           END ELSE
        21200                           IF PARM[OPCODE]=FXST OR (PARM[TYPE]=SI AND PARM[OPCODE]=SISTR) THEN
        21250                           !IF IT IS A STORE INTO MEMORY,IT HAS BEEN SENT TO THE SDB;
B53     21300                           BEGIN
        21350                               !IF NON-DATA-FLOW SYSTEM,RELEASE THE DECODER;
        21400                               IF FIXARCH\=DATAFLOW THEN NEXTGUY(FXIUCH,FXIUFREE);
        21450                               !AFTER A POSSIBLE DATA-DEPENDENCY WAIT,SEND OFF A MEMREF TO DO THE STORE.;
        21500                               IF \SST[PARM[SNKMAS]] THEN LOUNGE(SSTCH[PARM[SNKMAS]]);
        21550                               FREESLOT(SNKMAS);
        21600                               SCHEDULE(NEW MEMREF(TIME,PARM[MEMADD],CURRENT),0); LOUNGE(NONE);
E53     21650                           END ELSE
        21700                           !IF IT NEEDS THE EXECUTION UNIT;
B54     21750                           BEGIN
        21800                               !IF THE EXECUTION UNIT IS NOT AVAILABLE,WAIT.;
        21850                               IF NOT FXEUFREE THEN LOUNGE(FXEUCH) ELSE FXEUFREE:=FALSE;
        21900                               !FOR PIPELINED SYSTEM,RELEASE THE DECODER;
        21950                               IF FIXARCH=PIPELINE THEN NEXTGUY(FXIUCH,FXIUFREE);
        22000                               !EXECUTE THE INSTRUCTION;
        22050                               SCHEDULE(CURRENT,EXEC[PARM[OPCODE]]);
        22100                               !RELEASE THE EXECUTION UNIT;
        22150                               NEXTGUY(FXEUCH,FXEUFREE);
        22200                               !FOR A SERIAL SYSTEM,RELEASE THE DECODER NOW;
        22250                               IF FIXARCH=SERIAL THEN NEXTGUY(FXIUCH,FXIUFREE);
        22300                               !IF A RESULT IS TO BE STORED INTO MEMORY;
        22350                               IF PARM[TYPE]=SI AND PARM[OPCODE]=FETSTR THEN
B55     22400                               BEGIN
        22450                                   SCHEDULE(NEW MEMREF(TIME,PARM[MEMADD],CURRENT),0); LOUNGE(NONE);
E55     22500                               END;
E54     22550                           END;
        22600                           !NOW THE SINK IS AVAILABLE;
        22650                           RELEASE(SSTCH[PARM[SNKSLV]],SST[PARM[SNKSLV]]);
        22700                       !END OF FIXED POINT UNIT;
E45     22750                       END ELSE
        22800
        22850                       !FLOATING POINT INSTRUCTIONS;
B56     22900                       BEGIN
        22950                           SCHEDULE(CURRENT,1); !TRANSFER TO FLOATING POINT UNIT;
        23000                           !IS THE FLOATING POINT DECODER FREE?;
        23050                           IF NOT FLIUFREE THEN LOUNGE(FLIUCH) ELSE FLIUFREE:=FALSE;
        23100                           SCHEDULE(CURRENT,1); !FLOATING POINT DECODING;
        23150                           !DOES IT NEED A REGISTER SOURCE OPERAND;
        23200                           IF PARM[SRCSLV]\=INDEP AND(PARM[TYPE]=RR OR (PARM[TYPE]=RX AND PARM[STORE]=1)) THEN
B57     23250                           BEGIN
        23300                               !SEND OFF AN OPERAND PROCESS TO GET THE REGISTER OPERAND.;
        23350                               OFFSPRING:-NEW OPERAND(TIME,CURRENT); SCHEDULE(OFFSPRING,0);
```

```
E57     23400                           END ELSE
        23450                           IF \MEMOP THEN
B58     23500                           BEGIN
        23550                               !IF IT IS A 1-OPERAND INSTRUCTION,HALF THE SYNCHRONIZATION IS OVER.;
        23600                               SYNC:=1; FREESLOT(SRCMAS);
E56     23650                           END;
        23700                           !DOES IT NEED A SINK OPERAND AS WELL?;
        23750                           IF NOT(PARM[SNKSLV]=INDEP OR PARM[OPCODE]=FLSPLD OR PARM[OPCODE]=FLLD OR
B59     23800                            PARM[OPCODE]=FLST) THEN BEGIN
        23850                               !IS THE SINK OPERAND AVAILABLE?;
        23900                               IF SST[PARM[SNKMAS]] THEN
B60     23950                               BEGIN
        24000                                   FREESLOT(SNKMAS); !FREE THE SINK SLOT IN THE SST.;
        24050                                   SCHEDULE(CURRENT,1); !OPERAND FORWARDING TIME;
        24100                                   !IF DATA-FLOW ARCHITECTURE,RELEASE DECODER;
        24150                                   IF FLARCH=DATAFLOW THEN NEXTGUY(FLIUCH,FLIUFREE);
E60     24200                               END ELSE
        24250                               !IF THE SINK OPERAND IS NOT AVAILABLE;
B61     24300                               BEGIN
        24350                                   !FOR DATA-FLOW ARCHITECTURE,RELEASE DECODER;
        24400                                   IF FLARCH=DATAFLOW THEN NEXTGUY(FLIUCH,FLIUFREE);
        24450                                   !WAIT FOR OPERAND TO BECOME AVAILABLE;
        24500                                   LOUNGE(SSTCH[PARM[SNKMAS]]); FREESLOT(SNKMAS);
        24550                                   !FOR NON-DATA-FLOW ARCHITECTURE,TRANSFER THE OPERAND;
        24600                                   IF FLARCH\=DATAFLOW THEN SCHEDULE(CURRENT,1);
E61     24650                               END;
        24700                               !IF IT IS A COMPARE INSTRUCTION, THE OERAND IS AVAILABLE NOW.;
        24750                               IF PARM[OPCODE]=FLCMP THEN RELEASE(SSTCH[PARM[SNKSLV]],SST[PARM[SNKSLV]]);
E59     24800                           END ELSE
B62     24850                           BEGIN
        24900                               IF PARM[OPCODE]\=FLST THEN FREESLOT(SNKMAS);
        24950                               IF FLARCH=DATAFLOW THEN NEXTGUY(FLIUCH,FLIUFREE);
E62     25000                           END;
        25050
        25100                           !SYNCHRONIZE WITH THE OPERAND PROCESS;
        25150                           SYNC:=SYNC+1;
        25200                           !IF THE 2 PATHS HAVE NOT YET MERGED,WAIT.;
        25250                           IF SYNC<2 THEN LOUNGE(NONE) ELSE
        25300                           IF OFFSPRING=/=NONE THEN SCHEDULE(OFFSPRING,0);
        25350                           !IF A MEMORY OPERAND IS NEEDED IN A NON-DATA-FLOW SYSTEM,FORWARDING TIME;
        25400                           IF FLARCH\=DATAFLOW AND MEMOP THEN SCHEDULE(CURRENT,1);
        25450                           IF PARM[OPCODE]=FLLD THEN
        25500                           !IF IT IS LOAD INTO A REGISTER,IT HAS BEEN DONE;
B63     25550                           BEGIN
        25600                               !FOR NON-DATA-FLOW SYSTEMS,RELEASE DECODER;
        25650                               IF FLARCH\=DATAFLOW THEN NEXTGUY(FLIUCH,FLIUFREE);
E63     25700                           END ELSE
        25750                           IF PARM[OPCODE]=FLST THEN
        25800                           !IF IT IS A STORE INTO MEMORY,IT HAS BEEN SENT OFF TO THE SDB;
B64     25850                           BEGIN
        25900                               !IF NON-DATA-FLOW SYSTEM,RELEASE THE DECODER;
        25950                               IF FLARCH\=DATAFLOW THEN NEXTGUY(FLIUCH,FLIUFREE);
```

```
        26000                                   !AFTER A POSSIBLE DATA-DEPENDENCY WAIT,SEND OFF A MEMREF TO DO THE STORE.;
        26050                                   IF \SST[PARM[SNKMAS]] THEN LOUNGE(SSTCH[PARM[SNKMAS]]);
        26100                                   FREESLOT(SNKMAS); !FREE THE SINK SLOT IN THE SST.;
        26150                                   SCHEDULE(NEW MEMREF(TIME,PARM[MEMADD],CURRENT),0); LOUNGE(NONE);
E64     26200                               END ELSE
        26250
        26300                               IF PARM[OPCODE]>=FLMUL THEN
        26350                               !FLOATING POINT MULTIPLIES AND DIVIDES;
B65     26400                               BEGIN
        26450                                   !IS THE MULTIPLY EXECUTION UNIT AVAILABLE?;
        26500                                   IF NOT FLMDFREE THEN LOUNGE(FLMDCH) ELSE FLMDFREE:=FALSE;
        26550                                   !FOR PIPELINED SYSTEM,RELEASE THE DECODER;
        26600                                   IF FLARCH=PIPELINE THEN NEXTGUY(FLIUCH,FLIUFREE);
        26650                                   !EXECUTE THE INSTRUCTION;
        26700                                   SCHEDULE(CURRENT,EXEC[PARM[OPCODE]]);
        26750                                   !RELEASE THE EXECUTION UNIT;
        26800                                   NEXTGUY(FLMDCH,FLMDFREE);
        26850                                   !FOR A SERIAL SYSTEM,RELEASE THE DECODER NOW;
        26900                                   IF FLARCH=SERIAL THEN NEXTGUY(FLIUCH,FLIUFREE);
        26950                               !END OF FLOATING POINT MULTIPLIES;
E65     27000                               END ELSE
        27050
        27100                               !IF IT NEEDS THE ADD EXECUTION UNIT;
B66     27150                               BEGIN
        27200                                   !IS THE ADD EXECUTION UNIT AVAILABLE?;
        27250                                   IF NOT FLADFREE THEN LOUNGE(FLADCH) ELSE FLADFREE:=FALSE;
        27300                                   !FOR PIPELINED SYSTEM,RELEASE THE DECODER;
        27350                                   IF FLARCH=PIPELINE THEN NEXTGUY(FLIUCH,FLIUFREE);
        27400                                   !1 CYCLE IN 1ST STAGE OF ADD PIPELINE;
        27450                                   SCHEDULE(CURRENT,1);
        27500                                   !RELEASE THE EXECUTION UNIT;
        27550                                   NEXTGUY(FLADCH,FLADFREE);
        27600                                   !COMPARES DO NOT NEED A RESULT TRANSFER CYCLE WHILE ADDS DO.;
        27650                                   SCHEDULE(CURRENT,(IF PARM[OPCODE]=FLCMP THEN 1 ELSE 2));
        27700                                   !FOR A SERIAL SYSTEM,RELEASE THE DECODER NOW;
        27750                                   IF FLARCH=SERIAL THEN NEXTGUY(FLIUCH,FLIUFREE);
E66     27800                               END;
        27850
        27900                               !NOW THE SINK IS AVAILABLE;
        27950                               RELEASE(SSTCH[PARM[SNKSLV]],SST[PARM[SNKSLV]]);
E56     28000                           END OF FLOATING POINT UNIT;
        28050                           !NOW SET THE CONDITION CODE AND RELEASE ALL INSTRUCTIONS WAITING FOR IT;
        28100                           IF PARM[CCTAG]=ACTIVETAG THEN RELEASE(BCONDCH,CONDCODE);
E41     28150                       END OF NON-BRANCHING INSTRUCTIONS;
        28200
        28250                       !INTER-TERMINATION STATISTICS FOR NORMAL INSTRUCTIONS;
        28300                       TIMESTATS(NTERM,TIME-LASTTERM); LASTTERM:=TIME;
E32     28350                   END OF NOT TRACEND;
E21     28400               END OF NOT ENDPROG;
E19     28450           END OF NOT-ABORTED-INSTRUCTIONS;
        28500
        28550           !TERMINATION STATISTICS FOR ALL INSTRUCTION PROCESSES;
```

```
        28600               TIMESTATS(ITERM,TIME-LASTINST); LASTINST:=TIME;
E15     28650           END OF INSTRUCTION;
        28700
        28750           CPUCONTROL CLASS OPERAND(PARENT);
        28800           !THE OPERAND PROCESS.IT IS CREATED BY AN INSTRUCTION PROCESS THAT NEEDS A SOURCE OPERAND;
        28850           REF(INSTRUCTION) PARENT; !THE PARENT INSTRUCTION;
B67     28900           BEGIN
        28950               INTEGER I,MASTER;
        29000
        29050               !COPY THE DESCRIPTOR ARRAY OF THE PARENT;
        29100               FOR I:=1 STEP 1 UNTIL 12 DO PARM[I]:=PARENT.PARM[I];
        29150
        29200               IF PARM[OPCODE]<FLOAT THEN
        29250               !FIXED POINT OPERAND;
B68     29300               BEGIN
        29350                   IF PARENT.MEMOP THEN
        29400                   !IF IT IS A MEMORY OPERAND;
B69     29450                   BEGIN
        29500                       !SI INSTRUCTIONS NEED SINKS FECTHED NOT SOURCES;
        29550                       MASTER:=IF PARM[TYPE]=SI THEN SNKMAS ELSE SRCMAS;
        29600                       !WAIT FOR THE OPERAND TO BECOME AVAILABLE.;
        29650                       IF \SST[PARM[MASTER]] THEN LOUNGE(SSTCH[PARM[MASTER]]); FREESLOT(MASTER);
        29700                       !SCHEDULE A MEMREF TO GET THE OPERAND;
        29750                       SCHEDULE(NEW MEMREF(TIME,PARM[MEMADD],CURRENT),0); LOUNGE(NONE);
        29800                       !FOR A DATA-FLOW SYSTEM,OPERAND CAN BE TRANSFERRED NOW ITSELF;
        29850                       IF FIXARCH=DATAFLOW THEN SCHEDULE(CURRENT,1);
        29900                       !FOR NON-SI INSTRUCTIONS,RELEASE THE SOURCE OPERAND LOCATION;
        29950                       IF PARM[TYPE]\=SI THEN RELEASE(SSTCH[PARM[SRCSLV]],SST[PARM[SRCSLV]]);
        30000                   !END OF FIXED MEMORY OPERAND;
E69     30050                   END ELSE
        30100
        30150                   !FIXED POINT REGISTER OPERAND;
B70     30200                   BEGIN
        30250                       IF \SST[PARM[SRCMAS]] THEN
        30300                       !IF IT IS NOT AVAILABLE;
B71     30350                       BEGIN
        30400                           LOUNGE(SSTCH[PARM[SRCMAS]]); FREESLOT(SRCMAS);
        30450                           !IF DATA-FLOW SYSTEM,FORWARDING IS DONE AUTOMATICALLY;
        30500                           IF FIXARCH\=DATAFLOW THEN SCHEDULE(CURRENT,1);
E71     30550                       END ELSE
B72     30600                       BEGIN
        30650                           FREESLOT(SRCMAS); SCHEDULE(CURRENT,1);
E72     30700                       END;
        30750                       !RELEASE THE OPERAND LOCATION;
        30800                       RELEASE(SSTCH[PARM[SRCSLV]],SST[PARM[SRCSLV]]);
E70     30850                   END OF FIXED REGISTER OPERAND;
        30900
        30950                   !SYNCHRONIZE WITH PARENT INSTRUCTION;
        31000                   PARENT.SYNC:=PARENT.SYNC+1;
        31050                   IF PARENT.SYNC<2 THEN LOUNGE(NONE) ELSE SCHEDULE(PARENT,0);
        31100               !END OF FIXED POINT OPERAND;
E68     31150               END ELSE
```

```
        31200
        31250               !FLOATING POINT OPERAND;
B73     31300               BEGIN
        31350                   IF PARENT.MEMOP THEN
        31400                   !IF IT IS A MEMORY OPERAND;
B74     31450                   BEGIN
        31500                       IF \SST[PARM[SRCMAS]] THEN LOUNGE(SSTCH[PARM[SRCMAS]]); FREESLOT(SRCMAS);
        31550                       !SCHEDULE A MEMREF TO GET THE OPERAND;
        31600                       SCHEDULE(NEW MEMREF(TIME,PARM[MEMADD],CURRENT),0); LOUNGE(NONE);
        31650                       !FOR A DATA-FLOW SYSTEM,OPERAND CAN BE TRANSFERRED NOW ITSELF;
        31700                       IF FLARCH=DATAFLOW THEN SCHEDULE(CURRENT,1);
E74     31750                   END ELSE
        31800
        31850                   !FLOATING POINT REGISTER OPERAND;
B75     31900                   BEGIN
        31950                       IF \SST[PARM[SRCMAS]] THEN
        32000                       !IF IT IS NOT AVAILABLE;
B76     32050                       BEGIN
        32100                           LOUNGE(SSTCH[PARM[SRCMAS]]); FREESLOT(SRCMAS);
        32150                           !IF DATA-FLOW SYSTEM,FORWARDING IS DONE AUTOMATICALLY;
        32200                           IF FLARCH\=DATAFLOW THEN SCHEDULE(CURRENT,1);
E76     32250                       END ELSE
B77     32300                       BEGIN
        32350                           !OPERAND TRANSFER TIME.;
        32400                           FREESLOT(SRCMAS); SCHEDULE(CURRENT,1);
E77     32450                       END;
E75     32500                   END OF FLOATING REGISTER OPERAND;;
        32550
        32600                   !RELEASE THIS OPERAND LOCATION;
        32650                   RELEASE(SSTCH[PARM[SRCSLV]],SST[PARM[SRCSLV]]);
        32700                   !SYNCHRONIZE WITH PARENT INSTRUCTION;
        32750                   PARENT.SYNC:=PARENT.SYNC+1;
        32800                   IF PARENT.SYNC<2 THEN LOUNGE(NONE) ELSE SCHEDULE(PARENT,0);
E73     32850               END OF FLOATING POINT OPERAND;
E67     32900           END OF OPERAND;
        32950
        33000           CONTROL CLASS MEMREF(BANK,PARENT);
        33050           !THE MEMORY REFERENCE HANDLING PROCESS.CREATED FOR BOTH INSTRUCTION AND DATA FETCHES AND STORES;
        33100           INTEGER BANK; REF(CPUCONTROL) PARENT;
        33150           !THE BANK REFERENCED AND THE PARENT THAT MADE THE REFERENCE.;
B78     33200           BEGIN
        33250               MEMCOUNT:=MEMCOUNT+1; !MEMORY INITIATION STATISTICS;
        33300
        33350               !ONE-CYCLE ADDRESS TRANSFER TO THE MEMORY UNIT;
        33400               SCHEDULE(CURRENT,1);
        33450               !IS THE REFERENCED BANK AVAILABLE?;
        33500               IF \BANKFREE[BANK] THEN
B79     33550               BEGIN
        33600                   LOUNGE(BANKCH[BANK]); SCHEDULE(CURRENT,1);
E79     33650               END ELSE BANKFREE[BANK]:=FALSE;
        33700               !AFTER 1 ACCESS TIME,THE PARENT CAN BE ASKED TO LEAVE;
        33750               SCHEDULE(CURRENT,ACCESS+1); SCHEDULE(PARENT,0); SCHEDULE(CURRENT,CYCLE-ACCESS-1);
```

```
        33800               !BUT RELEASE THE BANK ONLY AFTER THE ENTIRE CYCLE TIME.;
        33850               NEXTGUY(BANKCH[BANK],BANKFREE[BANK]);
E78     33900           END OF MEMREF;
        33950
        34000           !INITIALIZE THE DESCRIPTOR VARIABLES INTRODUCED FOR DOCUMENTATION PURPOSES;
        34050           INDEP:=451;
        34100           !ARCHITECTURAL DESCRIPTORS;
        34150           SERIAL:=1; PIPELINE:=2; DATAFLOW:=3;
        34200           !INTER-EVEVT TIME DESCRIPTORS;
        34250           IGEN:=1; ITERM:=2; NTERM:=3;
        34300           !INSTRUCTION DESCRIPTORS;
        34350           TYPE:=1; OPCODE:=2; STORE:=3; CCTAG:=BRADD:=4; SNKMAS:=5; SRCMAS:=6;
        34400           SNKSLV:=7; SRCSLV:=8; INDREG:=9; MEMADD:=BRDIST:=11; TAKEBR:=12;
        34450           !INSTRUCTION TYPE DESCRIPTORS;
        34500           RR:=0; RX:=1; SI:=3; BRANCH:=5; TRACEND:=-1; FLOAT:=50;
        34550           !INSTRUCTION OPCODE DESCRIPTORS;
        34600           SVC:=9; BALR:=2; BC15:=5; BCR15:=6; BCT:=10;
        34650           SISTR:=1; FETSTR:=2; FETCH:=3;
        34700           FXLD:=1; FXAD:=2; FXCMP:=3; FXMUL:=4; FXDIV:=5; FXLA:=6; FXST:=7;
        34750           FLLD:=50; FLSPLD:=51; FLCMP:=53; FLMUL:=54; FLDIV:=55; FLST:=56;
        34800
        34850           LOOPSIZE:=4*BUFFERS; !THE UPPER LIMIT FOR LOOPMODE DECISIONS;
        34900           !ALL RESOURCES ARE FREE;
        34950           FXIUFREE:=FXEUFREE:=FLIUFREE:=FLADFREE:=FLMDFREE:=CONDCODE:=TRUE; ENDPROG:=FALSE;
        35000           !GENERATE WAITING CHAINS FOR THE RESOURCES;
        35050           IBUFCH:-NEW HEAD; FXIUCH:-NEW HEAD; FXEUCH:-NEW HEAD; FLIUCH:-NEW HEAD;
        35100           FLADCH:-NEW HEAD; FLMDCH:-NEW HEAD; BCONDCH:-NEW HEAD;
        35150           !GENERATE WAITING CHAINS FOR DATA DEPENDENCY WAITERS;
        35200           FOR I:=1 STEP 1 UNTIL INDEP DO
B80     35250           BEGIN
        35300               SST[I]:=TRUE; SSTCH[I]:-NEW HEAD;
E80     35350           END;
        35400           !GENERATE WAITING CHAINS FOR MEMORY CONFLICT WAITERS;
        35450           FOR I:=1 STEP 1 UNTIL BANKS DO
B81     35500           BEGIN
        35550               BANKFREE[I]:=TRUE; BANKCH[I]:-NEW HEAD;
E81     35600           END;
        35650           !INITIALIZE THE INSTRUCTION EXECUTION TIME TABLE;
        35700           EXEC[FXSPLD]:=2; EXEC[FXAD]:=2; EXEC[FXDIV]:=37; EXEC[FLMUL]:=3;
        35750           EXEC[FXCMP]:=1; EXEC[FXMUL]:=12; EXEC[FLDIV]:=11;
        35800
        35850           !INITIATE ENOUGH INSTRUCTION FETCHES TO FILL THE INSTRUCTION BUFFER;
        35900           BUFCOUNTER:=1; PC:=1;
        35950           FOR I:=1 STEP 1 UNTIL BUFFERS DO
B82     36000           BEGIN
        36050               ACTIVATE NEW INSTRUCTION(TIME+I-1,I) DELAY I-1;
        36100               PC:=1+MOD(PC,BANKS);
E82     36150           END;
        36200           !NOW GO AWAY AND LET ME SIMULATE IN PEACE;
        36250           PASSIVATE;
        36300
        36350           !OUTPUT THE SIMULATION STATISTICS.;
```

```
        36400           INSPECT OUTPUT DO
B83     36450           BEGIN
        36500               OUTTEXT("MEMORY CYCLE TIME:"); OUTINT(CYCLE,4); OUTIMAGE;
        36550               OUTTEXT("NUMBER OF MEMORY BANKS:"); OUTINT(BANKS,4); OUTIMAGE;
        36600               OUTTEXT("INSTRUCTION BUFFER SIZE:"); OUTINT(BUFFERS,4); OUTIMAGE;
        36650               OUTTEXT("FIXED POINT UNIT ARCHITECTURE: ");
        36700               OUTTEXT(IF FIXARCH=1 THEN "1 (SERIAL)" ELSE IF FIXARCH=2 THEN "2 (PIPELINED)  ELSE "3 (DATAFLOW)");
        36725               OUTIMAGE;
        36750               OUTTEXT("FLOATING POINT UNIT ARCHITECTURE: ");
        36800               OUTTEXT(IF FLARCH=1 THEN "1 (SERIAL)" ELSE IF FLARCH=2 THEN "2 (PIPELINED)" ELSE "3 (DATAFLOW)");
        36825               OUTIMAGE;
        36850               OUTTEXT("LOOPMODE: "); OUTTEXT(IF BRARCH THEN "1 (ON)" ELSE "0 (OFF)"); OUTIMAGE; OUTIMAGE;
        36900               OUTTEXT("EXECUTION TIME:"); OUTINT(TIME,6); OUTIMAGE;
        36950               OUTTEXT("NORMAL INSTRUCTION PROCESSES:"); OUTINT(NORMGEN,6); OUTIMAGE;
        37000               OUTTEXT("LOOPMODE INSTRUCTION PROCESSES:"); OUTINT(LOOPGEN,6); OUTIMAGE;
        37050               OUTTEXT("ABORTED INSTRUCTION PROCESSES:"); OUTINT(ABTERM,6); OUTIMAGE;
        37100               OUTTEXT("DUMMY INSTRUCTION PROCESSES:"); OUTINT(DUMTERM,6); OUTIMAGE;
        37150               OUTTEXT("MEMORY REFERENCES:"); OUTINT(MEMCOUNT,6); OUTIMAGE; OUTIMAGE;
        37200               OUTTEXT("SYSTEM PROCESS THRUPUT:"); OUTFIX(TIMEHIST[IGEN,0]/TIMEHIST[IGEN,1],6,10); OUTIMAGE;
        37250               OUTTEXT("SYSTEM INSTRUCTION THRUPUT:"); OUTFIX(TIMEHIST[ITERM,0]/TIMEHIST[ITERM,1],6,10); OUTIMAGE;
        37300               OUTTEXT("USEFUL INSTRUCTION THRUPUT:"); OUTFIX(TIMEHIST[NTERM,0]/TIMEHIST[NTERM,1],6,10); OUTIMAGE;
        37350               OUTIMAGE; OUTTEXT("GLICH:"); OUTINT(GLICH,5);
        37400               OUTTEXT("  GLICH TIME:"); OUTINT(GLTIME,7); OUTIMAGE;
        37450               CLOSE;
E83     37500           END OF INSPECT;
E2      37550       END OF SIMULATION;
        37600       INPUT.CLOSE;
E1      37650   END OF TDMOD;

DEFAULT SWITCHES USED

NO ERRORS DETECTED
```

LIST OF REFERENCES

[AND67a] Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo, "The IBM System 360/Model 91 : Machine Philosophy and Instruction Handling," IBM J. of Res. and Dev., Vol. 11, pp. 8-24, January 1967.

[AND67b] Anderson, S. F., J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System 360/Model 91: Floating - Point Execution Unit," IBM J. of Res. and Dev., Vol. 11, pp. 34-53, January 1967.

[BAL72] Ballance, R. S., J. A. Cocke, and H. G. Kolsky, "The Lookahead Unit," in Planning a Computer System, McGraw-Hill, 1962.

[BEL71] Bell, C. G. and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, 1971.

[BHA76] Bhandarkar, D. P., "A Hierarchy of Analytic Models for Complex Computer Systems,"The European Computing Conference on Computer Systems Evaluation, September 1976.

[BIR73] Birtwistle, G., et al., SIMULA Begin, Auerbach, 1973.

[BIR74] Birtwistle, G. and J. Palme, SIMULA Language Handbook - Part I, DECUS Program Library, 1974.

[BOL67] Boland, L. T., G. D. Granito, A. V. Marcotte, B. V. Messina, and J. W. Smith, "The IBM System 360/Model 91: Storage System," IBM J. of Res. and Dev., Vol. 11, pp. 54-68, January 1967.

[BRO72] Browne, J. C., K. M. Chandy, R. M. Brown, T. W. Keller, D. F. Towsley, and C. W. Dissly, "Hierarchical Techniques for the Development of Realistic Models of Complex Computer Systems," Proc. of the IEEE, Vol. 63, pp. 966-975, June 1975.

[DEN74] Dennis, J. B and D. P. Misunas, "A Preliminary Architecture for a Basic Data-flow Processor," Project MAC Computation Structures Group Memo 102, M.I.T., August 1974.

[DRA66] Draper, N. R. and H. Smith, Applied Regression Analysis, John Wiley, 1966.

[GRE69] Gregory, R. T. and D. L. Karney, A Collection of Matrices for Testing Computational Algorithms, Wiley-Interscience, 1969.

[IMS75] IMSL Library 2, Edition 5, International Mathematical and Statistical Libraries, Inc., 1975.

[KUM76a] Kumar, B., "Performance Evaluation of a Highly Concurrent Computer by Deterministic Simulation," Coordinated Science Laboratory Report R-717, University of Illinois, February 1976.

[KUM76b] Kumar, B. and E. S. Davidson, "Performance Evaluation of Highly Concurrent Computers by Simulation," submitted for publication in the Comm. ACM.

[SEK72] Sekino, A., "Performance Evaluation of Multiprogrammed Time-Shared Computer Systems," Project MAC Tech. Rep. 103, M.I.T., September 1971.

[SVO76] Svobodova, L., Computer Performance Measurement and Evaluation Methods: Analysis and Applications, Elsevier, 1976.

[TOM67] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Execution Units," IBM J. of Res. and Dev., Vol. 11, pp. 25-33, January 1967.

[TJA70] Tjaden, G. S. and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," IEEE - TC, Vol. C-19, pp. 889-895, October 1970.

[TSA72] Tsao, F. T., L. W. Comeau, and B. H. Margolin, "A Multi-factor Paging Experiment," Statistical Computer Performance Evaluation, Academic Press, 1972.

[ZEI76] Zeigler, B. P., Theory of Modelling and Simulation, Wiley-Interscience, 1976.

VITA

Balasubramanian Kumar was born in Pudukkottai, India on January 30, 1951. He received a B. Tech. degree in Electrical Engineering (Electronics) from the Indian Institute of Technology, Madras, India in 1973. At the Indian Institute of Technology, he received the President of India Prize for the best academic record in all branches of engineering in the graduating class of 1973. In 1976 he received an M.S. degree in Computer Science from the University of Illinois at Urbana-Champaign. From 1973 to 1977, he was employed as a graduate research assistant at the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign.